

Wind River® TIPC for VxWorks® 6

PROGRAMMER'S GUIDE

1.7

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, the Wind River logo, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:
installDir\product_name\3rd_party_licensor_notice.pdf.

Corporate Headquarters

Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

<http://www.windriver.com>

For information on how to contact Customer Support, please visit the following URL:

<http://www.windriver.com/support>

Contents

1	Introduction	1
1.1	Introduction	1
1.2	TIPC Overview	2
1.3	Comparison with Open-Source TIPC for Linux	3
1.4	Interoperability with Other Releases	3
1.5	Organization of This Document	4
2	TIPC Fundamentals	5
2.1	Introduction	5
2.2	TIPC Network Structure	6
2.2.1	Network Addresses	7
2.2.2	Supported Media for Communication Over Links	8
2.2.3	Multiple Links for Load-Sharing and Switchover	8
2.3	Messaging Overview	9
2.4	Message Reliability and Rejected Messages	11
2.5	TIPC Addressing	12

2.5.1	Network Address	12
2.5.2	Physical Addressing	13
2.5.3	Functional Addressing	13
2.5.4	Address Resolution	15
2.6	Multicasting	17
2.7	Subscriptions	18
3	Building VxWorks to Include Wind River TIPC	19
3.1	Introduction	19
3.2	Wind River TIPC Build Components	20
3.2.1	TIPC Footprint Reduction	27
3.2.2	TIPC socket API Build Component	30
3.2.3	TIPC memory pool Build Component	31
3.2.4	TIPC Media Types	32
	Ethernet Communication	33
	Shared Memory Communication	33
	Communication Using Distributed Shared Memory (DSHM)	35
3.2.5	TIPC network stack only	37
	Debugging TIPC on a Target System Built with the TIPC network stack only Component	38
	Including WDB Agent Proxy for TIPC in a VxWorks Build	39
	Including the WDB Target Agent in a Build with the TIPC Network Stack	40
	Starting the Target Server for TIPC Communication	40
3.2.6	TIPC configuration and display routines Build Component	42
3.2.7	Setting TIPC System Values	42
3.2.8	TIPC prioritized interfaces Build Component	45
3.3	Configuring Wind River TIPC	46
3.3.1	Setting Parameters in the TIPC Configuration String	48
3.3.2	Setting the be (bearer) Parameter	51

3.3.3	Accessing the Configuration String from the VxWorks Boot Loader	53
3.3.4	Implementing tipcConfigInfoGet()	53
3.4	Building VxWorks from Workbench	55
4	Using tipcConfig to Configure and Monitor TIPC	63
4.1	Introduction	63
4.2	tipcConfig Syntax and Command Options	64
4.2.1	Constraints on the Ordering of Command Options in tipcConfig Commands	75
4.2.2	The -be Command Option	75
4.2.3	Specifying a Domain	77
4.2.4	The -dest Command Option	81
4.2.5	Sample Log Output	83
4.2.6	Sample Output for the “ls” (Link Statistics) Option	83
4.2.7	Remote Management	84
	tipcConfig Command Options available for remote management	84
	Enabling Remote Management	84
	The -dest Command Option for Specifying the Address of a Node to be Managed	85
4.2.8	Using the -netid Option to Set Up Separate TIPC Networks Within a LAN	86
4.2.9	The -nt Command Option	86
	Sample Output for the -nt Command Option	87
4.2.10	Sample Output for the -p (Ports) Option	89
5	Subscriptions	91
5.1	Introduction	91
5.2	Creating and Using the TIPC Subscription Service	92
5.2.1	Creating a Subscription	92

5.2.2	Receiving a Subscription Event Notification	93
6	Using the Wind River VxWorks Simulator with TIPC	95
6.1	Introduction	95
6.2	Simulating a Standalone TIPC Node	96
6.3	Simulating a Network of TIPC Nodes	96
6.3.1	Simulating a Network of TIPC Nodes That Use Ethernet	96
6.3.2	Simulating a Network of TIPC Nodes That Use Shared Memory	98
6.3.3	Simulating a Network of TIPC Nodes That Use DSHM	99
7	Using Wind River System Viewer with TIPC	101
7.1	Introduction	101
7.2	TIPC Events Covered by System Viewer	102
7.3	Event Levels	102
7.4	Including TIPC System Viewer Instrumentation in a VxWorks Image Project	104
7.4.1	Building TIPC with the Network Stack	104
7.4.2	Building TIPC without the Network Stack	105
8	Using the TIPC Test Suite	107
8.1	Introduction	107
8.2	Including the Test Suite in a Project	108
8.3	Running Tests in the Test Suite	108
8.3.1	The tipcTS Shell Command	109
8.3.2	The tipcTC shell Command	109
8.3.3	Tests in the TIPC Test Suite	110

8.4	Sample Output	111
9	TIPC Native API	115
9.1	Introduction	115
9.2	Differences Between Using the Socket API and the Native API	116
9.3	Callback Routines	118
9.4	Structures for Handling Message Data	121
9.5	Routines in the TIPC Native API	121
9.6	Examples	123
9.6.1	Performing Basic Port Operations	123
9.6.2	Registering a TIPC User	124
9.6.3	Receiving a Synchronous Message	125
9.6.4	Using the TIPC Topology Service	126
A	Libraries	131
B	Socket and Utility Routines	147
C	TIPC Native Routines	177
D	Header File Definitions	213
D.1	Introduction	213
D.2	Definitions	214
E	Sample TIPC Application	219
E.1	Introduction	219
E.2	TIPC Inventory Simulation	220

E.2.1	Description	220
E.2.2	Source Code	222
F	TIPC Log Messages	247
F.1	Introduction	247
F.2	Log Messages	248
	TIPC info Messages	248
	TIPC warning Messages	248
	TIPC error Messages	250
	Index	253

1

Introduction

- 1.1 Introduction 1
- 1.2 TIPC Overview 2
- 1.3 Comparison with Open-Source TIPC for Linux 3
- 1.4 Interoperability with Other Releases 3
- 1.5 Organization of This Document 4

1.1 Introduction

The Transparent Inter Process Communication (TIPC) protocol is a network protocol that allows applications to communicate easily and efficiently in both single node environments and environments containing clusters of nodes. TIPC was originally developed at Ericsson in the 1990s.

The TIPC Project home page, at <http://tipc.sourceforge.net>, provides downloads, additional documentation about TIPC, technical support, and a TIPC discussion list.

The current release of Wind River TIPC is a port of the open-source Linux 1.7.5 implementation of TIPC, with some minor enhancements and bug fixes. Wind River TIPC 1.7 and open-source Linux 1.7.5 can interoperate within a network.

The TIPC protocol specification currently exists only in draft form and is subject to change. At the time of this writing, the most recent version of the specification available was **draft-spec-tipc-02.txt**, dated May 15, 2006 (see <http://tipc.sourceforge.net/documentation.html>).

1.2 TIPC Overview

This section provides a brief overview of TIPC. For more detailed information, see [2. TIPC Fundamentals](#).

When describing multiprocessing computing environments, the term *cluster* is often used to denote a group of interconnected computers that work together as a single computer. Such clusters can be made up of a large number of computers and can accommodate changes in topology as computers are added to, or drop out of, a cluster. In contrast to the Internet, the computers within a cluster are usually interconnected in such a manner that a message can get to its destination in a single hop.

TIPC has been expressly developed to meet the needs of applications running within a cluster. Its main features include:

- a location-transparent addressing scheme that makes services within clustered computers appear to belong to a single computer
- rapid, reliable interprocess communication within a node and between nodes, using either connection-oriented or connectionless modes of operation
- rapid notification of changes in topology and the ability to adjust quickly to these changes

Both Wind River TIPC and the open-source Linux implementation of TIPC provide access to TIPC capabilities through the well-known socket API, with some TIPC-specific modifications. In addition, TIPC provides a separate *native* API that can provide a smaller footprint and faster performance than the socket API. Wind River TIPC supports the native API for kernel applications, only. RTPs must use the TIPC socket API. In a kernel application, you can make calls to both the socket API and the native API.¹

1. Note that the TIPC native API has not been finalized by the TIPC Working Group of the Multicore Association (see <http://www.multicore-association.org>) and is still subject to change.

1.3 Comparison with Open-Source TIPC for Linux

Wind River TIPC 1.7 supports all features of TIPC that are available with open-source Linux TIPC 1.7.5 and is fully interoperable with it.

Wind River TIPC supports the following capabilities which are not available with open-source Linux TIPC 1.7.x:

- Communication between nodes through shared memory and distributed shared memory (DSHM) (see [3.2.4 TIPC Media Types](#), p.32).
- Prioritization of interfaces that use HEND drivers (see [3.2.8 TIPC prioritized interfaces Build Component](#), p.45; a HEND driver is a device driver that follows the Hierarchical Enhanced Network Driver (HEND) design introduced in VxWorks 6.2)

In addition to the differences in capabilities listed above, Wind River TIPC defines the TIPC socket-address structure (**sockaddr_tipc**) differently than Linux TIPC does. Wind River TIPC's definition follows the Berkeley Software Distribution (BSD) convention of an 8-bit length field and an 8-bit field for address family, rather than the single 16-bit field for address family used by Linux TIPC. For Wind River TIPC's **sockaddr_tipc** definition, see [D. Header File Definitions](#).

1.4 Interoperability with Other Releases

Wind River TIPC 1.7 is not interoperable with open-source Linux releases 1.7.1 and 1.7.2, because it uses an improved name and route-distribution algorithm that is incompatible with them. It is interoperable with open-source Linux release 1.7.3 and later.

TIPC 1.5 and 1.6 only support a single cluster in a network. Aside from new features in Wind River TIPC 1.7, Wind River TIPC 1.7 is interoperable with TIPC 1.5 and 1.6, with the following restriction: Nodes with TIPC 1.5 or TIPC 1.6 can only communicate with TIPC 1.7 nodes in the same cluster.

1.5 Organization of This Document

The remaining chapters in this book are organized as follows:

- [2. TIPC Fundamentals](#) provides basic technical information about TIPC and its implementation in Wind River TIPC.
- [3. Building VxWorks to Include Wind River TIPC](#) describes how to include Wind River TIPC binaries in a build of VxWorks.
- [4. Using `tipcConfig` to Configure and Monitor TIPC](#) describes how to use the **tipcConfig** utility to dynamically set and monitor TIPC configuration parameters.
- [5. Subscriptions](#) describes the TIPC subscription facility that makes it possible for nodes to learn about the availability of services throughout the network.
- [6. Using the Wind River VxWorks Simulator with TIPC](#) describes how to use the VxWorks target simulator, Wind River VxWorks Simulator, with Wind River TIPC.
- [7. Using Wind River System Viewer with TIPC](#) describes how to use System Viewer to display and log TIPC socket events.
- [8. Using the TIPC Test Suite](#) describes how to use the TIPC test suite to make sure that communication between nodes is working correctly.
- [9. TIPC Native API](#) describes the TIPC native API and provides examples of its usage.
- [A. Libraries](#) describes the Wind River TIPC library files and lists their public routines.
- [B. Socket and Utility Routines](#) describes the routines in the TIPC socket API. In addition, it describes the TIPC configuration and Show routines.
- [C. TIPC Native Routines](#) describes the routines in the TIPC native API.
- [D. Header File Definitions](#) gives the public **#define** and structure definitions in the **tipc.h** header file for Wind River TIPC.
- [E. Sample TIPC Application](#) provides a sample application that illustrates the use of the Wind River TIPC socket API and added utility routines.
- [F. TIPC Log Messages](#) lists the messages that can appear in the TIPC log.

2

TIPC Fundamentals

2.1	Introduction	5
2.2	TIPC Network Structure	6
2.3	Messaging Overview	9
2.4	Message Reliability and Rejected Messages	11
2.5	TIPC Addressing	12
2.6	Multicasting	17
2.7	Subscriptions	18

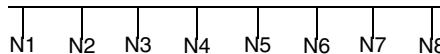
2.1 Introduction

This chapter provides a basic introduction to both the TIPC protocol and Wind River TIPC. All routines mentioned in this chapter are more thoroughly described in *B. Socket and Utility Routines*.

2.2 TIPC Network Structure

A TIPC network is a hierarchical structure superimposed on a physical network. A TIPC network is made up of one or more *zones*. Each zone contains one or more *clusters*. A cluster is made up of individual computers, or *nodes*. The nodes in a cluster are connected by *links*. In a TIPC network, a *link* is a logical connection between nodes. TIPC creates direct links between all nodes in a cluster, resulting in a (logical) full mesh topology. The links between nodes require an underlying physical network that makes communication between the nodes possible. For example, consider the following physical network with eight nodes:

Figure 2-1 **Physical Network with Eight Nodes**



Given the configuration in [Figure 2-1](#), TIPC allows you to define zones and clusters in any way you wish, as long as no node is included in more than one cluster or more than one zone. For example, any of the following TIPC networks would be possible:

- One zone containing a single cluster that consists of nodes N1, N4, and N7.
- One zone containing two clusters. Cluster one consists of nodes N1, N2, and N5; cluster two consists of nodes N3, N4, N7, and N8.
- Two zones containing one cluster each. Zone one contains a cluster consisting of nodes N1, N2, N3; zone two contains a cluster consisting of all the remaining nodes.

As the examples above indicate, not all physical connections between nodes need to be links in a TIPC network and the topology of a TIPC network can be very different from the physical network that underlies it.

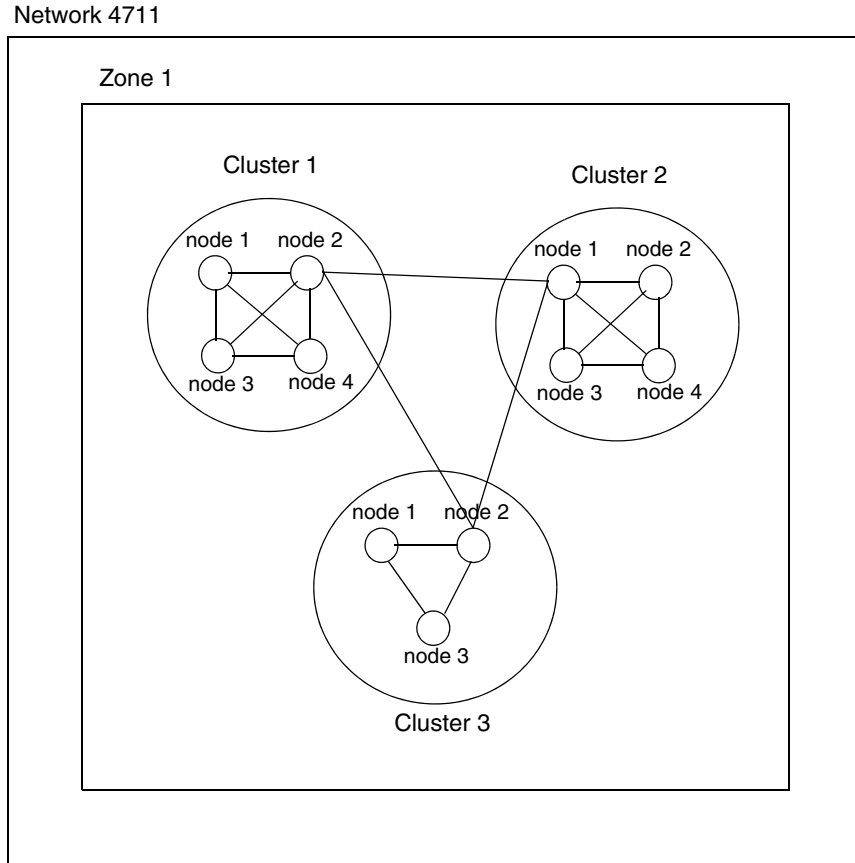
Typically, clusters are made up of nodes that share a common location, such as a shelf or room. Each cluster in a zone has direct links to all other clusters in the zone. Each zone has direct links to all other zones in a network. A direct link between two zones or between two clusters is established when a node in one zone or cluster has a direct link to a node in another zone or cluster.

Currently, the TIPC specification also provides for *secondary nodes*. A secondary node is only required to have a link to one other node in a cluster. The use of secondary nodes is under review and may be dropped from the specification.

Neither Wind River TIPC nor the open-source Linux version of TIPC supports secondary nodes.

Figure 2-2 illustrates a TIPC network containing a single zone with three clusters.

Figure 2-2 **Wind River TIPC Network Topology**



2.2.1 Network Addresses

Every TIPC network has a network ID. Each node within a network is assigned a unique network address that indicates its zone, cluster, and node number. Each of these values is an integer ranging from 1 to the maximum value specified for the network. A network address is usually denoted using the syntax $\langle Z.C.N \rangle$, as in

<1.1.5> for zone 1, cluster 1, and node 5. The TIPC protocol specification sets the following limits on the allowable number of zones within a networks, clusters within a zone, and nodes within a cluster,

Zones within a network	255
Clusters within a zone	4095
Nodes within a cluster	4095

To simplify the management of networks, TIPC employs a network discovery mechanism that only requires each node to have a network address (zone, cluster, node number) and each bearer on a node to have a *domain* (see [4.2.3 Specifying a Domain](#), p.77). Only nodes within a bearer's domain can establish links with the bearer's node. Beyond this, the nodes in a TIPC network automatically discover one another and establish links.

2.2.2 Supported Media for Communication Over Links

Currently, Wind River TIPC supports communication between nodes using Ethernet and through both shared memory and distributed shared memory (DSHM). For more information on the communication options, see [3.2.4 TIPC Media Types](#), p.32.

Nodes in a network can communicate over more than one type of medium; the only requirement is that every node in a cluster must be able to communicate with every other node in the cluster and that there is a direct link between each cluster in a zone and each zone in a network. For example, all the nodes in a cluster can have Ethernet links to each other, half the nodes in the cluster can have additional shared-memory links to each other, and a quarter of the nodes can have additional distributed shared memory links to each other.

To control the way TIPC creates links between nodes, clusters, and zones, when you specify a bearer and media type for a node, you also specify the a domain, and the domain determines the links that can be established.

2.2.3 Multiple Links for Load-Sharing and Switchover

If there are duplicate links of equal priority (see the explanation of *priority* under [3.3.2 Setting the be \(bearer\) Parameter](#), p.51) between two nodes, TIPC automatically provides load sharing over the links. In addition, if one link in a pair fails, the other link handles communication for it. The duplicate links do not need to use the same medium.

If you have duplicate physical networks, you can define a TIPC network in which each node in a cluster has duplicate links to all other nodes in the cluster, with all links having the same priority. In this case, TIPC automatically provides load sharing between all links and switchover for all links.

As long as the underlying physical connections support it, you can have up to eight links, with any combination of media, from one node to another. The following rules summarize load-sharing and switchover with multiple links between nodes:

- If one link is set to a higher priority than all other links, TIPC routes all traffic across it.
- If two links share the highest priority setting, TIPC shares the traffic between them.

No more than two links can have the same priority setting (see [3.3.2 Setting the *be \(bearer\) Parameter*](#), p.51).

- If duplicate links share the highest priority and one of them fails, the other takes over all traffic.
- If one link has a higher priority than all others and it fails, the link next in priority takes over for it.
- Load-sharing and switchover apply across media.

For example, if there is only an Ethernet link and a shared-memory link between two nodes, and both links are assigned the same priority, TIPC shares traffic between them, and when one fails, the other handles all traffic.

2.3 Messaging Overview

All communication within a TIPC network involves an exchange of data between *ports*. A port may send data to itself, to another port on the same node, or to a port that lies on another node in the network. A port can send data to multiple ports through TIPC multicasting (see [2.6 Multicasting](#), p.17). Multicasting is only supported for connectionless sockets.

User applications can create and use ports in a Wind River TIPC network through a conventional socket API, with some TIPC-specific extensions. In addition, a separate native API (see [9. TIPC Native API](#)) is available for TIPC programming.

Some ports are also created and used by Wind River TIPC, itself. A TIPC port is automatically created whenever an application creates a TIPC socket by calling **socket()** and specifying the **AF_TIPC** address family. The application sends and receives data through the socket, and TIPC routes the data through the associated port. When the socket is closed, the associated TIPC port is deleted.

Data is sent through a TIPC socket in units called *messages*. A message is a byte-array that can be from 1 to 66000 bytes long. The internal structure of the message is determined by the application. A byte stream, connection-oriented option, **SOCK_STREAM**, is also available for stream data

Wind River TIPC allows messages to be exchanged in a reliable, connectionless manner using the **SOCK_RDM** socket type. It also supports the **SOCK_DGRAM** socket type, which is essentially the same as **SOCK_RDM** but does not guarantee reliable message delivery. Applications can send messages using **send()**, **sendmsg**, or **sendto()**; applications can receive messages using **recv()**, **recvfrom()** or **recvmsg()**.

Wind River TIPC allows messages to be exchanged over a reliable connection using the **SOCK_SEQPACKET** and **SOCK_STREAM** socket types. Two means of establishing a connection are provided:

- An application can issue an explicit connection request using **connect()**. Once the connection is accepted, it can send and receive messages using **send()**.
- An application can issue an *implied* connection request by sending a message using **sendto()** without using **connect()**. The connection is recognized as completed when a response is received using **recv()**, **recvfrom()**, or **recvmsg()**.

This approach establishes a connection without requiring the connecting ports to exchange handshaking messages prior to the exchange of data. It provides the performance of connectionless data transfer and, since it is connection-oriented, still guarantees a correlation between request and response. An implicit connection request is particularly suitable when a client needs to make a single request over a connection rather than a series of requests.

To allow a socket to receive connection requests, an application first calls **listen()**. Then, to handle connection requests, the application calls **accept()**. **accept()** waits for a connection request and when it receives one, whether the request is explicit or implicit, it creates a new socket that is connected to the requesting socket.

Once a connection is established, messages are typically exchanged using **send()** and **recv()** until one side terminates the connection by closing a socket.

2.4 Message Reliability and Rejected Messages

Wind River TIPC makes considerable effort to successfully deliver a message to its destination. For example, if a message is given to a Wind River TIPC link for transmission to another node and then discarded by the underlying network medium, the link detects the loss and retransmits the message. However, in some cases, a message is undeliverable. This can occur because:

- The specified destination does not exist.
- The message was delivered to the specified destination, but the associated socket was closed before it was received by the application.
- The message was sent to a socket (or a node) that had too many unreceived messages already in queue.

When Wind River TIPC is unable to deliver a message, it “rejects” it.

- If a socket configured for connectionless but reliable message transfer (**SOCK_RDM** or **SOCK_DGRAM**, with the **TIPC_DEST_DROPPABLE** flag unset (see **setsockopt()** in [B. Socket and Utility Routines](#))) sends undeliverable messages, Wind River TIPC returns the first 1024 bytes of each message to the originating socket.



NOTE: For all routines referred to in this chapter, see [B. Socket and Utility Routines](#) for syntax statements and descriptions.

- If a socket configured for connection-oriented transfer (**SOCK_STREAM** or, if **TIPC_DEST_DROPPABLE** (see **setsockopt()**) is not specified, **SOCK_SEQPACKET**) sends undeliverable messages, TIPC marks the first undeliverable message sent to the destination as rejected and returns the initial 1024 bytes of the message. Subsequent undeliverable messages to the same destination are discarded, since returning the first undelivered message signals the socket that the connection is not functional.
- If a socket is not configured for reliable message transfer, TIPC simply discards undeliverable messages without informing the sender.

To receive rejected messages, an application can call the **recvmsg()** routine.

To decrease the likelihood that an important message is rejected because the destination end is congested with unreceived messages, TIPC associates an *importance level* with the messages sent by a socket. You can assign an importance level by calling **setsockopt()** with its **optname** parameter set to **TIPC_IMPORTANCE**. There are four importance levels:

- **TIPC_LOW_IMPORTANCE** (default)
Messages are rejected at the first sign of congestion.
- **TIPC_MEDIUM_IMPORTANCE**
Messages are rejected at medium levels of congestion.
- **TIPC_HIGH_IMPORTANCE**
Messages are rejected only at high levels of congestion.
- **TIPC_CRITICAL_IMPORTANCE**
messages are never rejected due to congestion.

2.5 TIPC Addressing

TIPC uses three different address forms within a network:

- network address
- physical port address
- functional port address

2.5.1 Network Address

As mentioned previously, a TIPC network address has the form $\langle Z.C.N \rangle$, where Z is zone, C is cluster, and N is node. Depending on its value, it can apply to a specific node, to any node within a cluster, or to any node within a zone. Network addresses containing a zero, have special interpretations:

- The network address $\langle Z.C.0 \rangle$ applies to any node within cluster C .
- The network address $\langle Z.0.0 \rangle$ applies to any node within zone Z .
- An isolated node that is not part of a TIPC network can be assigned a network address of $\langle 0.0.0 \rangle$. This prevents TIPC from allocating resources for inter-node communication that will never be utilized.

Network address $\langle 0.0.0 \rangle$ can also be used as a *lookup domain* (see [2.5.4 Address Resolution](#), p.15) rather than a node address.

Both Wind River TIPC and the open-source Linux version of TIPC interpret network address <0.0.0> in the same way, which differs from the interpretation in the draft TIPC specification.

An application specifies a network address using a 32-bit integer made up of three fields:

- Zone – 8 bits
- Cluster – 12 bits
- Node – 12 bits

Wind River TIPC provides the following APIs to allow applications to easily manipulate network addresses:

- **tipc_addr()** – takes separate values for zone, cluster, and node and combines them into a TIPC address.
- **tipc_node()** – takes a TIPC address and returns the node number.
- **tipc_cluster()** – takes a TIPC address and returns the cluster number.
- **tipc_zone()** – takes a TIPC address and returns the zone number.

2.5.2 Physical Addressing

A TIPC port has a unique *port identity*, which is automatically generated by TIPC when the port is created. A port ID is made up of two components, the network address of the node containing the port, in <Z.C.N> format followed by a colon, and a 32-bit randomly generated reference value. Together, the two components constitute the physical address of the port. The following is an example:

<1.1.7:1086734332>

An application can determine the port ID associated with a TIPC socket using the **getsockname()** routine.

2.5.3 Functional Addressing

Most TIPC applications use functional addresses rather than port IDs when communicating with TIPC sockets. A functional address allows the application to exchange data with the desired socket without having to know the physical location of the socket within the network. This makes it easy for applications to continue processing even when the ports within a network are being created,

destroyed, or relocated dynamically; it also makes it easier for applications to perform load sharing between multiple ports.

TIPC provides two forms of functional addressing: *port name* and *port name sequence*.

A port name is made up of two 32-bit integers: a *type* and an *instance*. Typically, the type field identifies the service provided by the port, while the instance identifies some aspect of the service. For example, type 100 may indicate a printer, while instance 500 of type 100 may identify a specific type of printer or the owner of the printer.

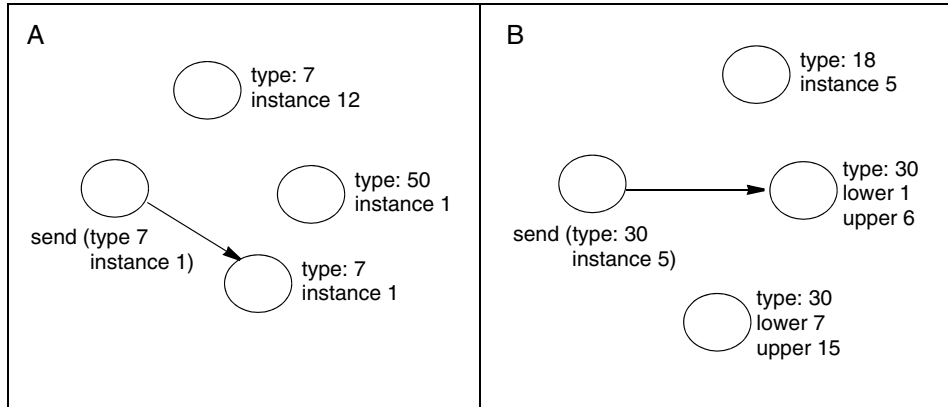
An application can assign a port name any type value within the 32-bit range, with the exception of types 0 through 63, which are reserved for TIPC-specific services.

Figure 2-3, section A, illustrates sending a request for a specific service (the port name) to the socket providing the service. In section A, all the destination sockets have port names assigned to them.

A port name sequence is made up of three 32-bit integers: a *type*, a *lower instance*, and an *upper instance*. It provides a way to specify a sequential set of port names using a single address, rather than having to specify the individual port names one at a time.

A port name sequence is typically assigned to a socket when the socket is capable of providing a series of related services. In Figure 2-3, section B illustrates sending a request for a specific service in a context in which some destination sockets have port name sequences assigned to them. Wind River TIPC allows a request to contain a port name, as in Figure 2-3, but not a port name sequence. The TIPC specification also allows port name sequences in requests for services.

Figure 2-3 Functional Addressing with Port Names and Port Name Sequences



To assign a functional address to a TIPC socket, an application calls **bind()** and specifies a service (the port's **type**) available through the socket and the scope within which the service is available. The scope can include all nodes within the socket's node, cluster, or zone. Following a bind operation, TIPC automatically sends (*publishes*) the new port-address information to all nodes within the designated scope.

A functional address remains bound to a socket until the socket is closed or the application calls **bind()** using a negative scope value (see the reference page for **bind()** in appendix [B. Socket and Utility Routines](#)). When the address is unbound, TIPC withdraws publication of the address from the network.

2.5.4 Address Resolution

TIPC allows an application to bind multiple port names or port name sequences to a single socket, unlike protocols such as TCP and UDP, which limit a socket to a single physical address. This capability is most commonly used when a socket is capable of providing multiple services.

TIPC also allows the same port name or port name sequence to be bound to multiple sockets in a network. This is useful when the network contains more than one socket that is capable of providing a given service.

TIPC does not allow port name sequences of a given type to have overlapping instance values, unless the ranges match exactly, as illustrated by the following examples:

Valid port name sequences (non-overlapping instances):

type 25 lower 1 upper 10	type 25 lower 1 upper 10	type 25 lower 11 upper 100	type 25 lower 1 upper 10	type 30 lower 5 upper 55
--------------------------------	--------------------------------	----------------------------------	--------------------------------	--------------------------------

Invalid port name sequences (overlapping instances):

type 25 lower 1 upper 10	type 25 lower 5 upper 15	type 25 lower 10 upper 10
--------------------------------	--------------------------------	---------------------------------

If a given port name is bound to multiple sockets, an application should only use a port name to request a service if it does not care what socket performs the service. To request a service from a specific socket, the application needs to use the socket's port ID.

When a connection request or a message is sent to a port name, TIPC resolves the name to a specific socket using the lookup domain specified by the message sender as part of the port address. The lookup domain is specified in <Z.C.N> format. Two name-resolution algorithms are available:

- If the lookup domain is <0.0.0>, TIPC uses a closest-first algorithm: it first tries to find a matching address belonging to a socket on the same node; if none exists, it then searches the node's cluster and finally the node's zone. If TIPC finds multiple matching addresses at a given level, it selects one in a round-robin manner. This approach gives preferential treatment to sockets that are "closer" to the sender.
- If the lookup domain is not <0.0.0>, TIPC uses a domain-search algorithm: it searches the specified network domain (node, cluster, or zone) to find a matching address. If TIPC finds multiple matching addresses at a given level, it selects one in a round-robin manner. Over time, this approach distributes the load throughout the available sockets in the specified domain.

2.6 Multicasting

When an application sends a message in a connectionless manner to a port name sequence, rather than a port name, TIPC sends a copy of the message to every port in the sender’s cluster that has a port name within the specified port name sequence. No more than a single copy of the message is sent to an individual port.

For example, suppose a message is sent in a connectionless manner to the following port name sequence:

{1000,100,200}

Each of the ports listed in [Table 2-1](#) will receive a single copy of the message.

Table 2-1 **Ports Receiving Multicast Messages for Port Name Sequence {1000,100,200}**

Port ID	Port Names/Port Name Sequences Bound to Port	Reason for Receipt of Message
<1.1.10:1234>	{1000,100}	Port name falls within the destination port name sequence.
<1.1.11:4321>	{1000,123}, {1000,175}	Both port names fall within the destination port name sequence.
<1.1.10:5678>	{1000,150}, {2000,150}	One of the port names falls within the destination port name sequence.
<1.1.12:5555>	1000,110,120}	All port names in the name sequence fall within the destination port name sequence.
<1.1.10:8888>	{1000,50,500}	Port name sequence contains all the port names in the destination port name sequence.
<1.1.14:9999>	{1000,170,300}	At least one port name in the port name sequence falls within the destination port name sequence.

None of the ports listed in [Table 2-2](#) will receive a copy of the message.

Table 2-2 Ports Not Receiving Multicast Messages for Port Name Sequence {1000,100,200}

Port ID	Port Names/Port Name Sequences Bound to Port	Reason for Non-Receipt of Message
<1.1.10:1111>	{2000,100,200}	Port-name types do not match.
<1.1.10:4444>	{1000,50,75}	No port names in the name sequence fall within the destination port name sequence.
<1.1.10:6666>	[None]	No port name to match on.

An application can multicasting messages for a single port name by sending the message to a port name sequence in which the upper and lower instance values are the same. For example, to multicast to ports that have port name {150,10} bound to them, the application can send a message to port name sequence {150,10,10}.

The following are restrictions on the use of multicasting:

- Messages are multicast only to nodes within the same cluster; there is no multicasting from one cluster to another.
- Messages must be sent in a connectionless manner.
- Messages must be sent to a port name sequence.
- The **domain** field of the **sockaddr_tipc** structure (see [D. Header File Definitions](#)) does not apply to multicasting.

When a message is sent to a functional address, the destination port name or port name sequence is specified in a **sockaddr_tipc** structure.

2.7 Subscriptions

TIPC provides a subscription service that allows applications to discover when specific services become available or unavailable. Using the subscription service, an application can subscribe to be notified about the availability of specific port names or port name sequences throughout the network. Any change in availability of the address associated with a subscription generates an event notification that the application can examine.

For information about creating and using TIPC subscriptions, see [5. Subscriptions](#).

3

Building VxWorks to Include Wind River TIPC

- 3.1 Introduction 19
- 3.2 Wind River TIPC Build Components 20
- 3.3 Configuring Wind River TIPC 46
- 3.4 Building VxWorks from Workbench 55

3.1 Introduction

To include TIPC in a VxWorks build, you need to create a VxWorks Image Project and include TIPC build components (see [3.2 Wind River TIPC Build Components](#), p.20). You can include TIPC build components using either Workbench or the **vxprj** command-line utility. For information on using Workbench to create a VxWorks Image Project and include build components, see the *Wind River Workbench User's Guide* and [3.4 Building VxWorks from Workbench](#), p.55. For information on using the **vxprj** command-line utility, see the *VxWorks Command-Line Tools User's Guide*.

An important feature of Wind River TIPC is that there are several build options that make it possible to reduce the size of the TIPC footprint by almost 40% and to reduce the combined size of VxWorks and TIPC on a target by 43%. For information on building TIPC with a reduced footprint, see [3.2.1 TIPC Footprint Reduction](#), p.27.

When you build VxWorks for TIPC, you can statically set a number of TIPC configuration parameters. You can set the same parameters dynamically, at startup and, additionally, you can dynamically set many of these and other parameters any time after startup using the **tipcConfig** utility. For more information on setting configuration parameters, see [3.3 Configuring Wind River TIPC](#), p.46 and [4. Using tipcConfig to Configure and Monitor TIPC](#).

3.2 Wind River TIPC Build Components

[Table 3-1](#) lists the build components for TIPC. To view the components as they appear in Workbench, see [Figure 3-2](#), under [3.4 Building VxWorks from Workbench](#), p.55. In [Table 3-1](#), required components are listed first.

Table 3-1 **Wind River TIPC Build Components**

Workbench	Macro	Description
TIPC	INCLUDE_TIPC	The core TIPC component, always required.
TIPC memory pool	INCLUDE_TIPC_MEMPOOL	A required component that allows you to allocate buffer space dedicated to TIPC sockets at startup. For further information, see 3.2.3 TIPC memory pool Build Component , p.31.
bootline configuration	INCLUDE_TIPC_CONFIG_HOOK_BOOT	<p>For dynamic configuration of TIPC parameters at startup time, choose either this component or the user configuration component (see the next table entry).</p> <p>Choosing this component instructs TIPC to get its configuration string from the other parameter of the VxWorks boot loader (see 3.3.3 Accessing the Configuration String from the VxWorks Boot Loader, p.53).</p> <p>You can include both dynamic and static configuration in the same build. You can also use the tipcConfig utility to configure and monitor many TIPC features dynamically, at startup or later. For more information, see 3.3 Configuring Wind River TIPC, p.46.</p>

Table 3-1 Wind River TIPC Build Components (cont'd)

Workbench	Macro	Description
user configuration	INCLUDE_TIPC_CONFIG_HOOK_USER	<p>For dynamic configuration of TIPC parameters at startup time, choose either this component or the bootline configuration component (see the preceding table entry).</p> <p>Choosing this component instructs TIPC to call the routine tipcConfigInfoGet(), for which you must provide a custom implementation (see 3.3.4 Implementing tipcConfigInfoGet(), p.53). If you implement tipcConfigInfoGet(), you must add the file containing the implementation to your VxWorks Image Project.</p> <p>You can include both dynamic and static configuration in the same build. You can also use the tipcConfig utility to configure and monitor many TIPC features dynamically, at startup or later. For more information, see 3.3 Configuring Wind River TIPC, p.46.</p>
Build TIPC from object library	INCLUDE_USE_LIBTIPC	<p>Builds TIPC from the precompiled object library, libtipc.a, which is provided when you install VxWorks. This component is included by default when you include TIPC (INCLUDE_TIPC) in your build. This component is automatically excluded when you build TIPC from source (see the next table entry).</p>

Table 3-1 Wind River TIPC Build Components (cont'd)

Workbench	Macro	Description
Build TIPC from source	INCLUDE_ BUILD_TIPC_ SRC	<p>Available only when TIPC source code is installed with VxWorks. For building TIPC with a reduced footprint, recompiles and builds TIPC from source code, based on the components you select for footprint reduction. This component is automatically included in your build when you include one or more of the components for footprint reduction. The components for footprint reduction are:</p> <ul style="list-style-type: none"> ▪ No TIPC debug (INCLUDE_TIPC_NODEBUG) ▪ No TIPC system messages (INCLUDE_TIPC_NOSYS_MSGS) ▪ No TIPC configuration service (INCLUDE_TIPC_NOCFG_SERVICE) ▪ No TIPC socket API (INCLUDE_TIPC_NOSOCKET) <p>For more information on TIPC footprint reduction, see the table entries for the listed components and 3.2.1 TIPC Footprint Reduction, p.27.</p>
TIPC and IP network stacks present	INCLUDE_ TIPC_IP	<p>Required for using the full VxWorks network stack. Choose either this component or the TIPC network stack only (INCLUDE_TIPC_ONLY) build component (see the next table entry).</p>

Table 3-1 **Wind River TIPC Build Components** (cont'd)

Workbench	Macro	Description
TIPC network stack only	INCLUDE_TIPC_ONLY	<p>Excludes services from the UDP/IP and TCP/IP protocols, which are not needed by TIPC. Choose either this component or the TIPC and IP network stacks present (INCLUDE_TIPC_IP) build component (see the preceding table entry).</p> <p>This build component includes only those network components required by TIPC and can significantly reduce the size of the VxWorks footprint. (For further information, see 3.2.5 TIPC network stack only, p.37 and 3.2.1 TIPC Footprint Reduction, p.27.)</p> <hr/> <p>NOTE: If you want to include TIPC network stack only in a Workbench build, you must first exclude the Network Components folder from your build (see Step 4 under 3.4 Building VxWorks from Workbench, p.55).</p> <hr/> <p>If you build the network stack with TIPC network stack only and want to use the WDB target agent to provide network communication between a TIPC target system and a host computer, you need to set up a WDB agent proxy on a target system. For further information, see Debugging TIPC on a Target System Built with the TIPC network stack only Component, p.38.</p>
Ethernet	INCLUDE_TIPC_MEDIA_ETH	For communication between nodes using Ethernet. In Workbench, this build component is included by default when you include TIPC. You can exclude it. In a command-line build, if you want Ethernet communication, you need to explicitly add INCLUDE_TIPC_MEDIA_ETH .
Shared Memory	INCLUDE_TIPC_MEDIA_SM	For communication between nodes through shared memory. Communication using shared memory is not available with all BSPs. For a list of the BSPs that support shared memory and additional information on TIPC shared memory, see 3.2.4 TIPC Media Types , p.32, and Shared Memory Communication , p.33.

Table 3-1 Wind River TIPC Build Components (cont'd)

Workbench	Macro	Description
DSHM Primary Interface	INCLUDE_DSHM_SVC_TIPC_PRIM	For communication between nodes using DSHM. Communication using DSHM is not available with all BSPs. For a list of the BSPs that support DSHM and additional information on DSHM for TIPC, see 3.2.4 TIPC Media Types , p.32 and Communication Using Distributed Shared Memory (DSHM) , p.35.
TIPC static configuration	INCLUDE_TIPC_CONFIG_STR	Allows configuration parameters to be set statically. You can include both dynamic and static configuration in the same build. You can also use the tipcConfig utility to configure and monitor many TIPC features dynamically, at startup or later. (For more information, see 3.3 Configuring Wind River TIPC , p.46.)
TIPC configuration and display routines	INCLUDE_TIPC_SHOW	Enables the use of the tipcConfig utility and TIPC command-line show routines. The tipcConfig utility allows you to dynamically configure TIPC features, to display current configuration settings, and to monitor the behavior of links and nodes in a TIPC network (see 4. Using tipcConfig to Configure and Monitor TIPC). The show routines provide information on TIPC memory allocation (see 3.2.6 TIPC configuration and display routines Build Component , p.42.)
No TIPC debug	INCLUDE_TIPC_NODEBUG	Excludes TIPC debug code from the build and reduces TIPC's footprint by approximately 17 KB (on a PPC32 target). For more information, see 3.2.1 TIPC Footprint Reduction , p.27. By default, TIPC debug code is included in the build; you need to explicitly exclude it.
No TIPC system messages	INCLUDE_TIPC_NOSYS_MSGS	Excludes TIPC system messages from the build and reduces TIPC's footprint by approximately 2 KB (on a PPC32 target). For more information, see 3.2.1 TIPC Footprint Reduction , p.27. By default, TIPC system messages are included in the build; you need to explicitly exclude them.

Table 3-1 Wind River TIPC Build Components (cont'd)

Workbench	Macro	Description
No TIPC configuration service	INCLUDE_TIPC_NOCFG_SERVICE	<p>Excludes code for the tipcConfig utility and for implementation of the following APIs that either set or display configuration values:</p> <ul style="list-style-type: none"> ▪ tipcConfig() ▪ tipcDataPoolShow() ▪ tipcSysPoolShow() <p>This reduces TIPC's footprint by approximately 17 KB (on a PPC32 target). For more information, see 3.2.1 TIPC Footprint Reduction, p.27.</p> <p>For information on the APIs that this build component excludes, see B. Socket and Utility Routines.</p>
No TIPC socket API	INCLUDE_TIPC_NOSOCKET	<p>(For kernel applications, only.) Excludes all code in support of the TIPC socket API from the build and reduces TIPC's footprint by approximately 12 KB (on a PPC32 target). For more information, see 3.2.1 TIPC Footprint Reduction, p.27.</p> <p>If you include this build component, you need to explicitly exclude the TIPC socket API (INCLUDE_CONFIG_TIPC_SOCKET_API) build component (see the next table entry).</p> <p>If you exclude the socket API, you need to use the TIPC native API (see 9. TIPC Native API).</p>

Table 3-1 Wind River TIPC Build Components (cont'd)

Workbench	Macro	Description
TIPC socket API	INCLUDE_CONFIG_TIPC_SOCKET_API	<p>If included, provides the following parameters for setting limits on the use of sockets:</p> <ul style="list-style-type: none"> ▪ Number of TIPC sockets (TIPC_NUM_SOCKETS) parameter allows you to specify the number of sockets that can be allocated on a node. ▪ The Socket Receive Queue Threshold (TIPC SOCK_RXQ_LIMIT) parameter allows you to set limits on the number of incoming messages that can be queued on TIPC sockets. <p>By default, this component is included in a TIPC build. If you include the No TIPC socket API (INCLUDE_TIPC_NOSOCKET) build parameter (see the preceding table entry), you need to explicitly exclude TIPC socket API (INCLUDE_CONFIG_TIPC_SOCKET_API).</p> <p>For further information, see 3.2.2 TIPC socket API Build Component, p.30.</p>
TIPC System Defines	INCLUDE_TIPC_DEFINES	<p>Allows you to set TIPC parameter values replacing initial system defaults. You can set the following:</p> <ul style="list-style-type: none"> ▪ Network ID ▪ Maximum values for the number of other nodes an individual node can have links to in its own cluster and in other clusters. ▪ Maximum values for ports on a node, subscriptions, and publications ▪ Status for remote management of a node—either enabled, or disabled <p>For detailed information on the parameters you can set, see 3.2.7 Setting TIPC System Values, p.42</p>

Table 3-1 Wind River TIPC Build Components (cont'd)

Workbench	Macro	Description
TIPC prioritized interfaces	INCLUDE_TIPC_HEND_INIT	<p>Available only with BSPs that are compatible HEND interfaces. A HEND interface is an interface that supports an HEND driver. For a list of HEND interfaces and the BSPs that are compatible with them, see 3.2.8 TIPC prioritized interfaces Build Component, p.45.</p> <p>This component does not appear in the Workbench Component Configuration Editor for projects that are not based on a qualifying BSP.</p> <p>TIPC prioritized interfaces (INCLUDE_TIPC_HEND_INIT) allows you to list interfaces that you want to give a higher priority for receiving packets than other interfaces. The interfaces listed must be configured exclusively for communication using TIPC; they cannot not be used for IP communication. For more information, see 3.2.8 TIPC prioritized interfaces Build Component, p.45.</p>
TIPC instrumentation	INCLUDE_WVTIPC	Enables use of Wind River System Viewer with TIPC (see 7. Using Wind River System Viewer with TIPC).
TIPC test suite demo	INCLUDE_TIPC_TS	Adds the TIPC test suite to the kernel image. For more information, see 8. Using the TIPC Test Suite .
TIPC inventory simulation demo	INCLUDE_TIPC_IS	Adds the TIPC inventory simulation, a sample application (see E. Sample TIPC Application), to a VxWorks kernel image. The code for the sample application is brought into the VxWorks Image Project as a kernel application residing in the project.

3.2.1 TIPC Footprint Reduction¹

In a build that does not make specific efforts to limit the size of Wind River TIPC's footprint, TIPC is likely to require approximately 125 KB of space.

You can reduce Wind River TIPC's footprint by close to 40% by excluding network services not required by TIPC from your build and by excluding individual

1. Estimates of footprint reduction in this section are based on the **wrSbc8560** BSP. Footprint reduction may be different for other BSPs.

features, such as debugging, that you may not need in a production build. [Table 3-2](#) lists build components and the reductions in footprint that you obtain by excluding them.

Table 3-2 TIPC Components and Footprint Reduction

Workbench Name	Macro	Footprint Reduction (in Kilobytes)	Comment
TIPC network stack only	INCLUDE_TIPC_ONLY	386	<p>For footprint reduction, you need to exclude the entire Network Components folder and then include this component. If you get the following error, it means that the network components were not excluded:</p> <p>incompatible with Boot parameter process (INCLUDE_NET_BOOT)</p> <p>For more information, see the table entry for TIPC network stack only in Table 3-1 under 3.2 Wind River TIPC Build Components, p.20).</p>
No TIPC debug	INCLUDE_TIPC_NODEBUG	17	Removes TIPC debug code from the build.
No TIPC system messages	INCLUDE_TIPC_NOSYS_MSGS	2	Removes code for TIPC system messages from the build.
No TIPC configuration service	INCLUDE_TIPC_NOCFG_SERVICE	17	<p>Removes code for the tipcConfig utility and the following APIs from the build:</p> <p>tipcConfig() tipcDataPoolShow() tipcSysPoolShow()</p> <p>For information on the tipcConfig utility, see 4. Using tipcConfig to Configure and Monitor TIPC; for information on individual APIs, see B. Socket and Utility Routines.</p>

Table 3-2 TIPC Components and Footprint Reduction (cont'd)

Workbench Name	Macro	Footprint Reduction (in Kilobytes)	Comment
No TIPC socket API	INCLUDE_TIPC_NOSOCKET	12	Excludes code all code in support of the TIPC socket from the build and reduces TIPC's footprint by approximately 12 Kilobytes (see 3.2.1 TIPC Footprint Reduction , p.27). If you include this build component, you need to explicitly exclude the TIPC socket API (INCLUDE_CONFIG_TIPC_SOCKET_API) build component (see the table entry for TIPC socket API in Table 3-1 under 3.2 Wind River TIPC Build Components , p.20).

- TIPC footprint, with all default components included: 125 KB.
- Total TIPC footprint reduction from the four **No TIPC** build components: 48 KB.
This is a 38% reduction in TIPC footprint.
- There is a 29% reduction in footprint when the TIPC socket API is retained but the other **No TIPC** build components are included.
- Default size of a VxWorks target with TIPC included: 1006 KB.
- Total VxWorks + TIPC footprint reduction from **TIPC network stack only** (**INCLUDE_TIPC_ONLY**): 386 KB
This is a 38% reduction in over-all footprint.
- Total VxWorks + TIPC footprint reduction from **TIPC network stack only** (**INCLUDE_TIPC_ONLY**) and all four **No TIPC** build components: 434 KB.
This is a 43% reduction in over-all footprint.

3.2.2 TIPC socket API Build Component

The **TIPC socket API** (`INCLUDE_CONFIG_TIPC_SOCKET_API`) build component is required in the current release and is automatically selected for inclusion when you include **TIPC**. **TIPC socket API** has the following parameters:

Table 3-3 **TIPC Socket API Parameters**

Parameter in Workbench	#define	Default value	Description
Number of TIPC sockets	TIPC_NUM_SOCKETS	200	The maximum number of concurrent sockets supported on a node. The space for the socket structures is allocated at startup.
Socket Receive Queue Threshold	TIPC_SOCKET_RXQ_LIMIT	2500	Sets limits on the number of incoming messages that can be queued on TIPC sockets. For more information, see Socket Receive Queue Threshold (TIPC_SOCKET_RXQ_LIMIT) Parameter , p.30.

Socket Receive Queue Threshold (TIPC_SOCKET_RXQ_LIMIT) Parameter

TIPC uses this parameter for managing traffic congestion. The parameter determines the maximum number of messages that can be queued on TIPC sockets, based on their importance level (see [2.4 Message Reliability and Rejected Messages](#), p.11 and, in [B. Socket and Utility Routines](#), **setsockopt()**.) In the current release, importance levels are handled as follows:

▪ **TIPC_LOW_IMPORTANCE**

For low-priority messages, maximum queue lengths for a single socket and all sockets on a node are:

- single socket: `TIPC_SOCKET_RXQ_LIMIT * 1`
- all sockets: `TIPC_SOCKET_RXQ_LIMIT * 2`

The default queue limit for a single socket is 2500; the queue limit across all sockets is 5000.

▪ **TIPC_MEDIUM_IMPORTANCE**

For medium-priority messages, maximum queue lengths for a single socket and all sockets on a node are:

- single socket: `TIPC_SOCK_RXQ_LIMIT * 2`
- all sockets: `TIPC_SOCK_RXQ_LIMIT * 4`

The default queue limit for a single socket is 5000; the queue limit across all sockets is 10000.

▪ **TIPC_HIGH_IMPORTANCE**

For high-priority messages, maximum queue lengths for a single socket and all sockets on a node are:

- single socket: `TIPC_SOCK_RXQ_LIMIT * 100`
- all sockets: `TIPC_SOCK_RXQ_LIMIT * 200`

The default queue limit for a single socket is 250000; the queue limit across all sockets is 500000.

▪ **TIPC_CRITICAL_IMPORTANCE**

The parameter value has no effect on critical messages. Critical messages are always queued, as long as buffer space is available.

3.2.3 TIPC memory pool Build Component

The **TIPC memory pool** (`INCLUDE_TIPC_MEMPOOL`) component allows you to specify the number and size of memory buffers specifically allocated at startup for use with TIPC sockets. Pre-allocating memory buffers can result in faster performance. If TIPC uses up its pre-allocated buffers, it can still call for additional buffer space from system memory.

When you include **TIPC memory pool**, you can configure sets of buffers, as listed in the following table.

Table 3-4 **TIPC Memory Pool Parameters**

Parameter in workbench	#define	Default Value
Number of 64 byte buffers	<code>TIPC_DATA_00064</code>	120
Number of 128 byte buffers	<code>TIPC_DATA_00128</code>	200

Table 3-4 **TIPC Memory Pool Parameters** (cont'd)

Parameter in workbench	#define	Default Value
Number of 256 byte buffers	TIPC_DATA_00256	40
Number of 512 byte buffers	TIPC_DATA_00512	40
Number of 1024 byte buffers	TIPC_DATA_01024	50
Number of 2048 byte buffers	TIPC_DATA_02048	20
Number of 4096 byte buffers	TIPC_DATA_04096	2
Number of 8192 byte buffers	TIPC_DATA_08192	0
Number of 16384 byte buffers	TIPC_DATA_16384	0
Number of 32768 byte buffers	TIPC_DATA_32768	0
Number of 65536 byte clusters	TIPC_DATA_65536	0

3.2.4 TIPC Media Types

Nodes in a Wind River TIPC network can communicate with each other using any combination of the following media types: Ethernet, shared memory, and distributed shared memory (DSHM). However, not all BSPs support either shared memory or DSHM and only Ethernet supports symmetrical multiprocessing (SMP).

To configure a node to use one or more media types, you need to include the appropriate build components and set the **be** (bearer) parameter in the TIPC configuration string (see [3.3.2 Setting the be \(bearer\) Parameter](#), p.51). The build components for TIPC media types are covered in the following sections:

- [Ethernet Communication](#), p.33
- [Shared Memory Communication](#), p.33
- [Communication Using Distributed Shared Memory \(DSHM\)](#), p.35

Ethernet Communication

In Workbench, Ethernet communication is included in a TIPC build by default, but you can exclude it from the build by deselecting the **Ethernet** (**INCLUDE_TIPC_MEDIA_ETH**) build component in the Component Configuration Editor.

Shared Memory Communication

If your BSP supports shared memory, you can build TIPC to communicate using shared memory. The following BSPs support the use of shared memory with TIPC:

- **mv5100**
- **cds8548**
- **hpcNet8641**
- **linux**
- **mv5100**
- **simpc**
- **solaris**
- **wrSbc8641d**

For shared memory, you need to include the **Shared Memory** (**INCLUDE_TIPC_MEDIA_SM**) build component and set TIPC shared-memory parameters.

VxWorks shared memory is set up in terms of a master board, whose local memory is used as shared memory, and slave boards, which can access the shared memory on the master board. The master board can also access its own shared memory. (For information on VxWorks shared memory, see the *VxWorks Kernel Programmer's Guide*.) When you set TIPC shared-memory parameters, you can set them to apply to master boards or to slave boards, as described in the following table.

Table 3-5 TIPC Shared-Memory Parameters

Workbench Description	Default Value
Macro Name Description	
Starting address of TIPC shared memory block SM_TIPC_ADRS	SM_TIPC_ADRS _DEFAULT
<p>(Master board, only) Address of the shared-memory pool on the master board. (If the board is a slave board, the value entered here is ignored.)</p> <p>The default setting, SM_TIPC_ADRS_DEFAULT, uses the default shared-memory address configured for the board support package (BSP) of the master board.</p> <p>In the case of a TIPC network consisting of multiple CPUs on a single board, you can allocate shared memory dynamically by setting SM_TIPC_ADRS to NONE.</p>	
Size of TIPC shared memory block SM_TIPC_SIZE	0
<p>(Master board, only) The size of the shared-memory pool, in bytes. (If the board is a slave board, the value entered here is ignored.)</p> <p>For the master board, in order to enable TIPC shared memory, you must change the default value of 0 to a value greater than zero. You can enter the value as SM_TIPC_SIZE_DEFAULT, which applies to all supported BSPs and is set at 0x00020000 (128K).</p>	
Shared memory packets size SM_TIPC_SM_PKT_SIZE	0 [equivalent to 2176 bytes]
<p>(Master board, only) The size of the packets used to contain shared-memory data, in bytes. A minimum packet size of 160 bytes is recommended.</p> <p>The default value of 0 is mapped to the VxWorks system default of 2176 bytes.</p> <p>If the board is a slave board, the value entered for SM_TIPC_SM_PKT_SIZE is ignored.</p>	

Table 3-5 TIPC Shared-Memory Parameters (cont'd)

Workbench Description	Default Value
Macro Name Description	
Number of buffers in the bearer pool SM_TIPC_NUM_BUF	60
(Master or slave board) The number of buffers of local memory to be allocated for receiving TIPC data from shared memory. The buffers are not freed until a user application reads the data. If large bursts of traffic are expected, this number should be high. If not, it can be reduced.	
If SM_TIPC_NUM_BUF is too big, buffer space is allocated and unused; if it is too small, messages may be rejected and need to be retransmitted.	
Maximum packets queued in SM SM_TIPC_PKT_Q_LEN	0 [equivalent to 200 packets]
(Master or slave board) The number of packets that can be queued in shared memory on the master board for this node.	
If SM_TIPC_PKT_Q_LEN is set too high, a large number of packets may be queued for this node, and there may be insufficient shared-memory space for queuing packets for other nodes. If SM_TIPC_PKT_Q_LEN is set too low, receipt of shared-memory data may be delayed.	
The default value of 0 is mapped to the VxWorks shared-memory default of 200 packets.	

Communication Using Distributed Shared Memory (DSHM)

For general information on using DSHM, see the *VxWorks Kernel Programmer's Guide: Distributed Shared Memory*.

If your BSP supports DSHM, you can build VxWorks to support TIPC communication using DSHM. The following BSPs support DSHM:

- **hpcNet8641**
- **linux**
- **sb1250**
- **sb1480**
- **simpc**
- **solaris**

To use DSHM with TIPC, you need to include the **Primary TIPC interface** (`INCLUDE_DSHM_SVC_TIPC_PRIM`) build component in your build. The build component provides the configuration parameters listed in [Table 3-6](#).

Table 3-6 Primary TIPC interface Build Component Configuration Parameters

Workbench Description Macro Name Description	Default Value & Data Type
Hardware bus <code>DSHM_SVC_TIPC_PRIM_HW</code> The name of the bus that the DSHM TIPC interface is on.	<p>"plb"</p> <p>char *</p>
requested size of SM buffer pool <code>DSHM_SVC_TIPC_PRIM_SZ_REQ_SM_POOL</code> The requested size of the shared-memory buffer pool to use with DSHM for TIPC.	<p>0x35000</p> <p>UINT</p>
minimal acceptable size of SM buffer pool <code>DSHM_SVC_TIPC_PRIM_SZ_MIN_SM_POOL</code> The minimum acceptable size of the shared-memory buffer pool to use with DSHM for TIPC.	<p>0x35000</p> <p>UINT</p>
size of buffers in SM <code>DSHM_SVC_TIPC_PRIM_SZ_SM_BUFFER</code> The size of the buffers to use for DSHM with TIPC.	<p>0x800</p> <p>UINT</p>
TIPC link window <code>DSHM_SVC_TIPC_PRIM_LINK_WINDOW</code> The window size for the link. Window size is the number of packets sent on the link that the node keeps in memory without needing to receive an acknowledgement from the recipient.	<p>0</p> <p>UINT</p>

Table 3-6 Primary TIPC interface Build Component Configuration Parameters (cont'd)

Workbench Description Macro Name Description	Default Value & Data Type
Broadcast buffers	0
DSHM_SVC_TIPC_PRIM_N_BCAST_ENTRIES	UINT
The Number of buffers to make concurrently available for sending broadcasts. If set to 0, replicast is used, instead of broadcast.	
All BSPs supported in the current release must use replicast. Do not change the default setting.	
Maximum buffers allocated per-peer	32
DSHM_SVC_TIPC_PRIM_MAX_BUF_ALLOC	UINT
The maximum number of buffers to allocate for TIPC DSHM communication with an individual node, for both incoming and outgoing traffic.	

3.2.5 TIPC network stack only

Wind River TIPC does not use any of the services provided by the UDP or TCP/IP protocols. If these services are not required by other applications on a node, you can reduce the size of the VxWorks footprint by building the network stack with the **TIPC network stack only (INCLUDE_TIPC_ONLY)** build component, which includes only those components needed by TIPC.

The **TIPC network stack only** build component is available only if you build VxWorks from Workbench or from the command line using the **vxprj** build tool. (For information on using the **vxprj** build tool, see the *VxWorks Command-Line Tools User's Guide: Working with Projects and Components*.)

To build the network stack with **TIPC network stack only**:

1. Exclude the entire **Network Components** bundle from your build.

By default, many network components are initially set for inclusion in a VxWorks Image Project. This removes them from the build.

2. Include the TIPC build components you want, including the **TIPC network stack only** build option.

TIPC network stack only builds the network stack with only those components required by TIPC.

For detailed instructions, see [3.4 Building VxWorks from Workbench](#), p.55.

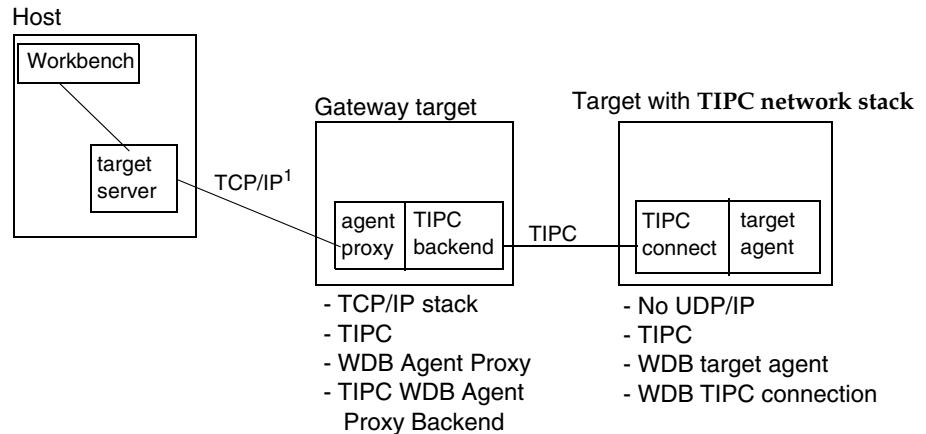
Building VxWorks and TIPC with **TIPC network stack only** may have implications for using Workbench debugging tools with a TIPC application that is running on a target system. This is described in the next section.

Debugging TIPC on a Target System Built with the TIPC network stack only Component

The VxWorks *WDB target agent* is a software component that resides on a target system and makes it possible to use Workbench debugging tools located on a host system with an application running on the target system (see *VxWorks Kernel Programmer's Guide: Target Tools*). Workbench communicates with the WDB target agent through a *target server* that runs on the host system.

If there is a full network stack on the target system, network communication between a target server and WDB target agent can be carried out using UDP/IP. However, if the target stack is built with **TIPC network stack only**, UDP/IP is not available. In this case, if the target needs to communicate with the host over a TIPC network, the WDB target agent and the target server cannot communicate directly and have to go through a proxy agent—*WDB Agent Proxy*. WDB Agent Proxy needs to reside in a target system separate from the target agent, as illustrated in [Figure 3-1](#).

Figure 3-1 Communication between a host and a target with a TIPC network stack build



1. Although the WDB target agent uses UDP in network communication with a target server, WDB Agent Proxy uses TCP/IP.

In Figure 3-1, WDB Agent Proxy TIPC (*TIPC backend* in the gateway target) and WDB TIPC connection (*TIPC connect* in the **TIPC network stack only** target) are required build components for TIPC communication.

The sections that follow describe how to build and initialize communication over a WDB agent proxy.

Including WDB Agent Proxy for TIPC in a VxWorks Build

To include the WDB agent proxy in a VxWorks build for a target system that will be used as a TIPC gateway, you need to include the following Workbench build components:

- **WDB Agent Proxy**
- **TIPC WDB Agent Proxy Backend**

Including the WDB Target Agent in a Build with the TIPC Network Stack

In Workbench, the simplest way to include the necessary WDB target-agent components for a build containing the **TIPC Network Stack** component is:

1. In the Workbench component tree, expand the **development tool components** bundle, right-click on **WDB agent components** and select **Include**.

The **Include** window for **WDB agent components** appears.

2. Accept all checked (default) components and, in addition, check **WDB TIPC connection**.
3. Click **Finish**.

For information on including WDB components in a build, see the *VxWorks Kernel Programmer's Guide: Target Tools*.

Starting the Target Server for TIPC Communication

You can start a target server and connect to a TIPC gateway for communication over a TIPC network from the Workbench target manager (see *Wind River Workbench User's Guide: New Target Server Connections*) or from the command line. The following command shows the TIPC command options you need to use in both cases:

```
tgtsvr -V -B wdbproxy -tipc -tgt targetTipcAddress -tipcpt tipcPortType -tipcpi  
tipcPortInstance wdbProxyIpAddress/name
```


Table 3-7 explains the italicized parameter values in the command:

Table 3-7 TIPC-Specific Parameter Values for Starting a Target Server

Parameter	Description
<i>targetTipcAddress</i>	The TIPC address of the target with the TIPC network stack in <Z.C.N> format. For example: <1.1.8>.
<i>tipcPortType</i>	<p>The TIPC port type (see 2.5.3 Functional Addressing, p.13) to use in connecting to the WDB target agent. The default port type for the connection is 70. You should accept the default port unless it is already in use.</p> <p>To change the port type:</p> <ul style="list-style-type: none"> ▪ In Workbench, change the WDB_TIPC_PORT_TYPE parameter under WDB TIPC connection ▪ For a command-line build, change the define for WDB_TIPC_PORT_TYPE, as in the following example: <pre>#undef WDB_TIPC_PORT_TYPE #define WDB_TIPC_PORT_TYPE 117</pre>
<i>tipcPortInstance</i>	<p>The TIPC port instance (see 2.5.3 Functional Addressing, p.13) to use in connecting to the WDB target agent. The default port instance for the connection is 71. You should accept the default port instance unless it is already in use.</p> <p>To change the port instance:</p> <ul style="list-style-type: none"> ▪ In Workbench, change the WDB_TIPC_PORT_INSTANCE parameter under WDB TIPC connection ▪ For a command-line build, change the define for WDB_TIPC_PORT_INSTANCE, as in the following example: <pre>#undef WDB_TIPC_PORT_INSTANCE #define WDB_TIPC_PORT_INSTANCE 118</pre>

Table 3-7 **TIPC-Specific Parameter Values for Starting a Target Server** (cont'd)

Parameter	Description
<i>wdbProxyIpAddress/name</i>	The IP address or DNS name of the gateway target running the WDB.

The following is a sample **tgtsvr** command for TIPC communication:

```
tgtsvr -V -B wdbproxy -tipc -tgt 1.1.8 -tipcpt 70 -tipcpi 71 192.168.1.5
```

3.2.6 **TIPC configuration and display routines Build Component**

The **TIPC configuration and display routines** component (**INCLUDE_TIPC_SHOW**) available only in VxWorks kernel mode, enables use of the **tipcConfig** utility (see [4. Using tipcConfig to Configure and Monitor TIPC](#)) and provides the following Show routines:

tipcDataPoolShow()

Displays statistics on the allocation and availability of clusters in the TIPC data pool.

tipcSysPoolShow()

Displays statistics on the allocation and availability of clusters in the TIPC system pool.

The Show routines are documented in [B. Socket and Utility Routines](#).

3.2.7 **Setting TIPC System Values**

The **TIPC System Defines** (**INCLUDE_TIPC_DEFINES**) build component allows you to set TIPC system values for the parameters listed in [Table 3-8](#).

Table 3-8 TIPC System Values Set through the TIPC System Defines Build Component

Workbench name	#define	Default Value	Description
Default Network ID	TIPC_DEF_NET_ID	4711	<p>Sets the default Network ID. The ID must be a value in the range from 1 to 9999.</p> <p>You can change the ID through the -netid option of the tipcConfig utility (see the entry for netid in Table 4-1 under 4.2 <i>tipcConfig Syntax and Command Options</i>, p.64.</p>
Max Ports	TIPC_DEF_MAX_PORTS	8191	<p>Sets the maximum number of ports that this node can create. The number should include both incoming and outgoing ports for services offered by the node and a small number of additional ports needed by TIPC for system purposes. The number of ports needed for system purposes can vary, but is generally less than ten.</p> <p>The number of ports must be a value in the range from 127 to 65535.</p> <p>Each node in a cluster can have a different setting for Max Ports (TIPC_DEF_MAX_PORTS).</p>
Max Nodes	TIPC_DEF_MAX_NODES	255	<p>Sets the maximum number of nodes that this node can have links to in its own cluster. This is also the highest node number (N) that can be used in a Z.C.N network address. Each node in a cluster should have the same Max Nodes (TIPC_DEF_MAX_NODES) setting. The setting must be a value in the range from 8 to 4095.</p>
Max Clusters	TIPC_DEF_MAX_CLUSTERS	8	<p>Sets the maximum number of links that this node can have to nodes in other clusters within its zone. The number must be a value in the range from 1 to 4095</p>
Max Remotes	TIPC_DEF_MAX_REMOTES	8	<p>Sets the maximum number of nodes that this node can have links to outside its own cluster. The number must be a value in the range from 0 to 255.</p>

Table 3-8 TIPC System Values Set through the TIPC System Defines Build Component (cont'd)

Workbench name	#define	Default Value	Description
Max Zones	TIPC_DEF_MAX_ZONES	4	Sets the maximum number of nodes that this node can connect to in other zones.
Remote Management	TIPC_DEF_REMOTE_MGT	1	<p>Sets the default value for remote management of the node. By default, remote management is enabled. To disable remote management by default, set this parameter to 0.</p> <p>When remote management of a node is enabled, other nodes in the network can use the tipcConfig utility to manage the node and display information about it. In the current release, remote management is limited to a subset of tipcConfig command options and only allows you to display information about a managed node. For more information, see 4.2.7 Remote Management, p.84.</p> <p>You can override the default setting through the -mng option of the tipcConfig utility (see the table entry for -mng, p.72).</p>
Max Publications	TIPC_DEF_MAX_PUBS	10000	<p>Sets the maximum number of services a node can maintain at one time.</p> <p>Max Publications (TIPC_DEF_MAX_PUBS) must be a value in the range from 1 to 65535.</p>
Max Subscriptions	TIPC_DEF_MAX_SUBS	2000	<p>Sets the maximum number of subscriptions that a node supports. This is equivalent to the maximum number of services that all applications on the node, combined, can subscribe to. For information on subscriptions, see 2.7 Subscriptions, p.18.</p> <p>Max Subscriptions (TIPC_DEF_MAX_SUBS) must be a value in the range from 1 to 65535.</p>

3.2.8 TIPC prioritized interfaces Build Component

The **TIPC prioritized interfaces (INCLUDE_TIPC_HEND_INIT)** build component is available only with BSPs that are compatible with HEND interfaces. A HEND interface is an interface that supports an HEND driver. The following interfaces support HEND drivers:

- **motTsec**
- **motEtsec**
- **motFec**
- **qeFcc**

The following BSPs are compatible with HEND interfaces:

ads8544	ads860	ads88x	cds8548
hpcNet864	m54x5evb	mds8360	pcPentium
pcPentium2	pcPentium3	pcPentium4	pcPentium_mp
wrSbc8540	wrSbc8560		

The **TIPC prioritized interfaces (INCLUDE_TIPC_HEND_INIT)** build component makes it possible to assign designated TIPC-only interfaces a higher priority for receiving packets than other interfaces. A TIPC-only interface is an interface used exclusively for handling TIPC. Other interfaces on a TIPC node can be configured to handle both TIPC and IP or IP only.

By default, packets received by a node are placed in a single queue and handled by the **tNet0** task. However, *prioritized* TIPC interfaces are queued separately from other interfaces and packets received by them are handled by a separate task, **tTpcRxTask**, that runs at a higher priority than **tNet0**. Thus, when a TIPC-only interface is prioritized, packets received by it are queued separately from IP packets and processed at a higher priority.

In Wind River TIPC 1.7, prioritized interfaces are restricted as follows:

- Only interfaces using an HEND driver can be prioritized.
An HEND driver is a device driver that follows the Hierarchical Enhanced Network Driver (HEND) design introduced in VxWorks 6.2. To assign an HEND driver to mottsec interfaces you need to include the **motTsecHend Hierarchical Enhanced Network Driver (INCLUDE_MOT_TSEC_HEND)** build parameter in your build.
- As noted earlier, only TIPC-only interfaces can be prioritized.



NOTE: There is no requirement that TIPC-only HEND interfaces be prioritized. On a single node you can, for example, prioritize one TIPC-only HEND interface and leave another TIPC-only HEND interface unprioritized.

- Prioritized interfaces cannot be reconfigured.

Once an image is built, a prioritized interface cannot be reconfigured at startup or later as a TIPC-plus-IP interface or an IP-only interface. Similarly, an interface that was not built as a prioritized interface cannot be reconfigured as a prioritized interface.

The **TIPC prioritized interfaces** (`INCLUDE_TIPC_HEND_INIT`) build component contains a single parameter, **TIPC interfaces using H-END** (`TIPC_HEND_CONFIG_STR`), for listing one or more interfaces that are to be prioritized. Interfaces are entered as a comma-separated list enclosed in quotes. The following is an example:

TIPC interfaces using H-END="mottsec1,mottsec2"

If you include the **TIPC prioritized interfaces** build component in your build, you also need to include the **motTsecHend Hierarchical Enhanced Network Driver** (`INCLUDE_MOT_TSEC_HEND`) build component. The path to the component in the Workbench component tree is:

**Hardware > device drivers >
motTsecHend Hierarchical Enhanced Network Driver**

3.3 Configuring Wind River TIPC

Depending on the TIPC components you include for your build, Wind River TIPC gets its initial parameter settings from a configuration string (see [3.3.1 Setting Parameters in the TIPC Configuration String](#), p.48) in one of the following ways:

- Statically, from a configuration string that is built into VxWorks (**TIPC static configuration**/`INCLUDE_TIPC_CONFIG_STR` build component; see [Table 3-1](#) under [3.2 Wind River TIPC Build Components](#), p.20).

- Dynamically, from a configuration string that is accessed through a routine called when VxWorks starts (the **bootline configuration**/INCLUDE_TIPC_CONFIG_HOOK_BOOT and **user configuration**/INCLUDE_TIPC_CONFIG_HOOK_USER build components; see Table 3-1 under 3.2 Wind River TIPC Build Components, p.20).
- From both a static configuration string and a dynamic string.

In addition, after startup, you can use the **tipcConfig** utility to dynamically configure and monitor many TIPC features (see 4. Using tipcConfig to Configure and Monitor TIPC).

Static configuration is most useful for parameters that have the same value across multiple nodes. An example of such a parameter is **max_nodes**, which sets the maximum number of nodes a given node can link to within its cluster. Dynamic configuration at startup is useful when nodes use a common VxWorks image but require different values for the same parameter. An example of this is the parameter **a** (node address, specified as <Z.C.N>), which has a unique value for every node.

If a given parameter is specified in both a dynamic and a static configuration string, the value in the dynamic configuration string takes precedence, except when specifying interfaces with the **be** parameter (see 3.3.1 Setting Parameters in the TIPC Configuration String, p.48). In the case of the **be** parameter, if you configure one interface in a dynamic configuration string and another interface in a static configuration string, both are valid.

To set Wind River TIPC to use static configuration, include the **TIPC static configuration** (INCLUDE_TIPC_CONFIG_STR) build component in your build (see Table 3-1 under 3.2 Wind River TIPC Build Components, p.20).

To set Wind River TIPC for dynamic configuration at startup, include one of the following parameters:

- **bootline configuration** (INCLUDE_TIPC_CONFIG_HOOK_BOOT)

In this case, to set the TIPC configuration string, you access it through the **other** parameter of the VxWorks boot loader (see 3.3.3 Accessing the Configuration String from the VxWorks Boot Loader, p.53).

- **user configuration** (INCLUDE_TIPC_CONFIG_HOOK_USER)

In this case, you need to implement the **tipcConfigInfoGet()** routine (see 3.3.4 Implementing tipcConfigInfoGet(), p.53). This option allows you to access the configuration string from a location of your choice. For example, you can

put the configuration string in a file and use `tipcConfigInfoGet()` to access the file on a local hard disk.

3.3.1 Setting Parameters in the TIPC Configuration String

When Wind River TIPC starts, it uses default parameter values, unless the values are specified in a TIPC configuration string. The configuration string is composed of a series of parameters and values, separated by semi-colons, as in the following example:

```
max_nodes=100;a=1.1.27;be=eth:cpm0
```

Table 3-9 lists the available configuration parameters. There are constraints on the order in which parameters can be specified. The ordering restrictions are:

```
max_ports > netid > a > be  
max_zones, max_nodes, max_clusters > a
```

where “>” means “precedes”.

For example, the following is a valid configuration string:

```
max_nodes=100;max_ports=200;netid=1000;a=1.1.27;log=1024;be=eth:cpm0
```

Configuration strings are case sensitive and cannot contain spaces.

Table 3-9 Configuration Parameters in the TIPC Configuration String

Parameter	Syntax	Default Value	Description
a [address]	a=Z.C.N	<0.0.0>	Sets network address of the node. If the node is not part of a network, set the address to <0.0.0>.
be [bearer]	<i>bearer_name</i> [<i>ldomain</i> [<i>lpriority</i>]]	N/A	Specifies the type of communication to use between nodes: Ethernet over a specific interface, shared memory, or both Ethernet and shared memory. In addition to specifying a medium, you can specify a <i>domain</i> and a priority to assign to communication with the node. For more information, see 3.3.2 Setting the be (bearer) Parameter , p.51. Before you enter the be parameter in a configuration string, you must enter the a (address) parameter.

Table 3-9 Configuration Parameters in the TIPC Configuration String (cont'd)

Parameter	Syntax	Default Value	Description
log	<code>log=size</code>	0	<p>The size, in bytes, of the log. A log size of 0 (the default) means that logging is turned off.</p> <p>If there is a log, the minimum size is 512 bytes. If you specify a log size less than the minimum (other than 0), the minimum size is used.</p> <p>If you enter a log size and later change the size, the log is reset to empty and the new size goes into effect.</p> <p>For an example of log output, see 4.2.5 Sample Log Output, p.83.</p>
max_clusters	<code>max_clusters=N</code>	8	<p>The maximum number of links that this node can have to nodes in other clusters within its zone.</p> <p>The number must be a value in the range from 1 to 4095</p>
max_nodes	<code>max_nodes=N</code>	255	<p>Sets the maximum number of nodes that this node can have links to in its own cluster. This is also the highest node number (N) that can be used in a Z.C.N network address. Each node in a cluster should have the same Max Nodes (TIPC_DEF_MAX_NODES) setting. The setting must be a value in the range from 8 to 4095.</p>
max_ports	<code>max_ports=N</code>	8191	<p>the maximum number of ports that his node can create. The number should include both incoming and outgoing ports for services offered by the node and a small number of additional ports needed by TIPC for system purposes. The number of ports needed for system purposes can vary, but is generally less than ten.</p> <p>The number of ports must be a value in the range from 127 to 65536.</p> <p>Each node in a cluster can have a different setting for max_ports.</p>

Table 3-9 Configuration Parameters in the TIPC Configuration String (cont'd)

Parameter	Syntax	Default Value	Description
log	log=size	0	<p>The size, in bytes, of the log. A log size of 0 (the default) means that logging is turned off.</p> <p>If there is a log, the minimum size is 512 bytes. If you specify a log size less than the minimum (other than 0), the minimum size is used.</p> <p>If you enter a log size and later change the size, the log is reset to empty and the new size goes into effect.</p> <p>For an example of log output, see 4.2.5 Sample Log Output, p.83.</p>
max_clusters	max_clusters=N	8	<p>The maximum number of links that this node can have to nodes in other clusters within its zone.</p> <p>The number must be a value in the range from 1 to 4095</p>
max_nodes	max_nodes=N	255	<p>Sets the maximum number of nodes that this node can have links to in its own cluster. This is also the highest node number (N) that can be used in a Z.C.N network address. Each node in a cluster should have the same Max Nodes (TIPC_DEF_MAX_NODES) setting. The setting must be a value in the range from 8 to 4095.</p>
max_ports	max_ports=N	8191	<p>the maximum number of ports that his node can create. The number should include both incoming and outgoing ports for services offered by the node and a small number of additional ports needed by TIPC for system purposes. The number of ports needed for system purposes can vary, but is generally less than ten.</p> <p>The number of ports must be a value in the range from 127 to 65536.</p> <p>Each node in a cluster can have a different setting for max_ports.</p>

Table 3-9 Configuration Parameters in the TIPC Configuration String (cont'd)

Parameter	Syntax	Default Value	Description
max_publ	max_publ= <i>N</i>	10000	Sets the maximum number of services a node can maintain at one time. Range: 1 to 65535.
max_subscr	max_subscr= <i>N</i>	2000	Sets the maximum number of subscriptions that a node supports. This is equivalent to the maximum number of services that all applications on the node, combined, can subscribe to. For information on subscriptions, see 2.7 Subscriptions , p.18. If <i>max_number</i> is specified, sets the maximum number of subscriptions the local node can have. Range: 1 to 65535.
max_zones	max_zones= <i>N</i>	4	The maximum number of nodes that this node can connect to in other zones.
netid	netid= <i>ID</i>	4711	The network ID used by the node. Range: 1 to 9999.

3.3.2 Setting the be (bearer) Parameter

The **be** parameter specifies the type of communication to use between nodes, either Ethernet, shared memory, or distributed shared memory (DSHM), and allows you to assign a *domain* and a priority to communication with a node.

There can be more than one instance of the **be** parameter in a configuration string. For example, you can use the parameter once to specify an Ethernet interface and a second time to specify the use of shared memory. You can also repeat the parameter to specify the use of multiple Ethernet interfaces. The current release supports up to eight active interfaces. Separate repeated uses of the parameter with semi-colons, as in the following example:

```
be=eth:fei1/1.1.0/12;be=sm:sm0/1.1.0/8
```

Before you configure the **be** parameter, you must always enter the **-a** parameter. The **-a** parameter sets the address of the node for which bearer information is set with the **-be** parameter.

The syntax for specifying the **be** parameter is:

bearer_name[/domain[/priority]]

where:

- *bearer_name* has one of the following formats:

- **eth:***interface_name* (Ethernet)
- **sm:sm0** (shared memory).

The **sm0** component is a fixed value and should not be modified.

Note that shared memory is not available with all BSPs (see [3.2.4 TIPC Media Types](#), p.32).

- **dshm:plb0** (distributed shared memory)

For the current release, the **plb0** component is a fixed value and should not be modified.

Note that DSHM is not available with all BSPs (see [3.2.4 TIPC Media Types](#), p.32).

- *domain* is an optional argument given in <Z.C.N> format that determines which nodes the current node can have links to.

In specifying a domain, a zero value for Z, C, or N, means that the domain includes all zones, clusters, or nodes:

- If *domain* is specified as <0.0.0>, all nodes in the network are included in the domain.
- If *domain* is specified as <1.0.0>, the domain is restricted to zone 1, but includes all nodes in all clusters within zone 1.
- If *domain* is specified as <Z.C.0>, where Z and C are the current node's zone and cluster, the domain includes only the nodes within the current node's cluster.

If no domain is specified, the current node can only have links to other nodes in its cluster. This is equivalent to a domain of Z.C.0, where Z is the node's own zone and C is the node's own cluster.

For more detailed information on domains, see [4.2.3 Specifying a Domain](#), p.77.

priority assigns a priority to communication over the specified interface, as follows:

- For priorities 0 through 31, the higher the integer, the greater the priority.

- A priority of 32 (the default) sets priority equal to the default priority of the medium (Ethernet or shared memory).
 - The default value for Ethernet is 10.
 - The default value for shared memory is 15.
 - The default value for distributed shared memory is 15.

3.3.3 Accessing the Configuration String from the VxWorks Boot Loader

To access the TIPC configuration string from the VxWorks boot loader:

1. Bring up the VxWorks boot prompt.

You can bring up the VxWorks boot prompt by booting, or rebooting, VxWorks and then pressing any key when you see the message:

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

2. At the VxWorks Boot prompt, enter **c** (for “change”):

```
[VxWorks Boot]:c
```

VxWorks prompts you to set configuration parameters. It displays one parameter at a time, showing a different parameter each time you press **Enter**.

3. Press **Enter** until you see the following prompt:

```
other :
```

4. Enter the configuration string. Do not enclose the string in double quotes.

3.3.4 Implementing `tipcConfigInfoGet()`

For dynamic configuration at startup, you can implement the routine `tipcConfigInfoGet()` to access the Wind River TIPC configuration string. This makes it possible to put the configuration string in a location where it is easy to modify, which is particularly useful when there are configuration parameters that are subject to frequent change or are different from one node to another.

`tipcConfigInfoGet()` has the following syntax:

```
STATUS tipcConfigInfoGet
(
    char * buffer,      /* buffer for null-terminated configuration string */
    UINT bufferlen      /* length of the buffer */
)
```

The **tipcConfigInfoGet()** routine returns **OK** on success, or **ERROR** on failure to place the configuration string in the designated buffer.

The file with the source code for **tipcConfigInfoGet()** must contain the following include:

```
#include <vxWorks.h>
```

If you implement **tipcConfigInfoGet()**, you must add the file containing the implementation to your VxWorks Image Project.

If you are calling **tipcConfigInfoGet()** for dynamic configuration of Wind River TIPC:

1. Place the source file containing **tipcConfigInfoGet()** in the build directory for your board-support package (BSP):

installDir/target/config/bspDir

2. Open the make file in the build directory and add the name of the object file that contains **tipcConfigInfoGet()** to the end of the line that starts with **MACH_EXTRA**, as in the following example:

```
MACH_EXTRA = sysSpeed.o mtxI2c.o hawkI2c.o sysASpeed.o  
tipcCfgInfoGet.o
```

3.4 Building VxWorks from Workbench

This section describes how to include Wind River TIPC in a Workbench VxWorks Image Project. For detailed information on using Workbench, see the *Wind River Workbench User's Guide*.

To include Wind River TIPC in a VxWorks Image Project:

1. Launch Workbench and open the workspace that contains your VxWorks Image Project. If you do not have an existing VxWorks Image Project, see the *Wind River Workbench User's Guide* for instructions on how to create one.
2. Expand your VxWorks Image Project and double-click **Kernel Configuration** to display the component tree for your project.
3. Expand the component folders as follows:

Network Components > Network Protocol Components > TIPC components

4. If you want to build VxWorks with a TIPC-specific network stack that excludes all UDP/IP and TCP/IP services:
 - a. Right-click **Network Components** and choose **Exclude**.
The **Exclude** window for network components appears.
 - b. Click **Finish** to exclude all network components from your build.

For information on the TIPC-specific network stack, see [3.2.5 TIPC network stack only](#), p.37.

In the next step, you include TIPC build components in your VxWorks Image Project. [Figure 3-2](#) shows the TIPC build components in the Workbench component tree.

Figure 3-2 TIPC Build Components in the Workbench Component Configuration Editor

Component Configuration	
Description	Name
[-] TIPC components	FOLDER_TIPC
[-] TIPC advanced configuration	FOLDER_TIPC_ADVANCED
[-] TIPC footprint reduction	FOLDER_TIPC_FOOTPRINT
No TIPC configuration service	INCLUDE_TIPC_NOCFG_SERVICE
No TIPC debug	INCLUDE_TIPC_NODEBUG
No TIPC socket API	INCLUDE_TIPC_NOSOCKET
No TIPC system messages	INCLUDE_TIPC_NOSYS_MSGS
[-] TIPC kernel demos	FOLDER_TIPC_DEMOS
TIPC inventory simulation demo	INCLUDE_TIPC_IS
TIPC test suite demo (default)	INCLUDE_TIPC_TS
[-] TIPC library selection	SELECT_TIPC_BUILD
Build TIPC from object library (default)	INCLUDE_USE_LIBTIPC
Build TIPC from source * [see note 1]	INCLUDE_BUILD_TIPC_SRC
[-] TIPC memory pool (default)	INCLUDE_TIPC_MEMPOOL
[-] TIPC prioritized interfaces * [see note 2]	INCLUDE_TIPC_HEND_INIT
[-] TIPC socket API (default)	INCLUDE_CONFIG_TIPC_SOCKET_API
[-] TIPC system defines	INCLUDE_TIPC_DEFINES
[-] TIPC initialization	SELECT_TIPC_INIT
TIPC dynamic configuration (default)	SELECT_TIPC_CONFIG_HOOK
Bootline configuration (default)	INCLUDE_TIPC_CONFIG_HOOK_BOOT
User configuration	INCLUDE_TIPC_CONFIG_HOOK_USER
[-] TIPC static configuration	INCLUDE_TIPC_CONFIG_STR
[-] TIPC media types	SELECT_TIPC_MEDIA_TYPES
DSHM primary interface * [see note 2]	INCLUDE_DSHM_SVC_TIPC_PRIM
Ethernet (default)	INCLUDE_TIPC_MEDIA_ETH
Shared Memory * [see note 2]	INCLUDE_TIPC_MEDIA_SM
[-] TIPC stack support	SELECT_TIPC_STACK
TIPC and IP network stacks present (default)	INCLUDE_TIPC_IP
TIPC network stack only	INCLUDE_TIPC_ONLY
TIPC (default)	INCLUDE_TIPC
TIPC configuration and display routines	INCLUDE_TIPC_SHOW

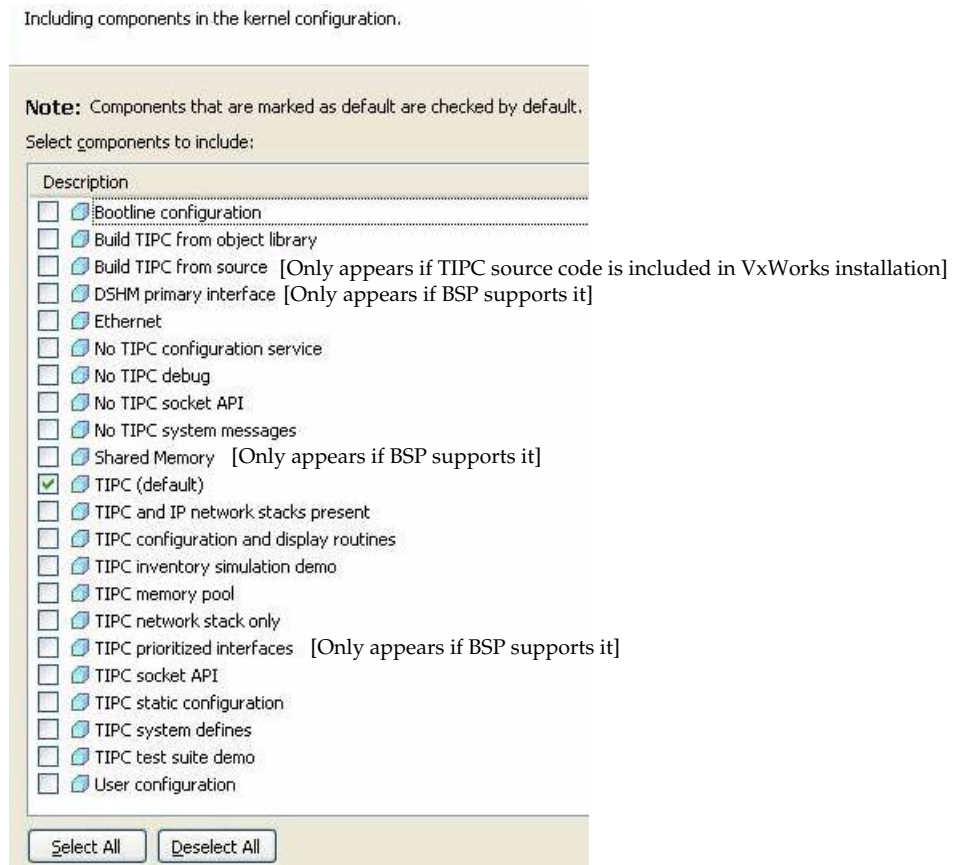
*1 Component only appears if TIPC source code is included in your installation

*2 Component only appears in the configuration tree if supported by the BSP used in the build

5. Right-click on **TIPC components** and Select **Include**

The Workbench **Include** window appears. It lists the available TIPC components.

Figure 3-3 TIPC components in the Workbench Include window



The **TIPC** component is included by default. When you include **TIPC** in a build, the following components are also included by default, and you do not need to check them:

- Bootline configuration
- Build TIPC from Object library
- Ethernet
- TIPC memory pool
- TIPC socket API
- TIPC and IP network stacks present
- TIPC system defines

6. For build components that require a choice among one or more alternatives, make your selections as described below. For information about individual components, see [3.2 Wind River TIPC Build Components](#), p.20.

Static and Dynamic Configuration

You can include either static configuration, dynamic configuration, or both (see [3.3 Configuring Wind River TIPC](#), p.46).

For Static configuration, include:

- **TIPC static configuration**

For dynamic configuration, include one of the following:

- **bootline configuration** (default)

Choosing this component instructs TIPC to get the configuration string from the **other** parameter of the VxWorks boot loader (see [3.3.3 Accessing the Configuration String from the VxWorks Boot Loader](#), p.53).

- **user configuration**

Choosing this component instructs TIPC to call the routine **tipcConfigInfoGet()**, for which you must provide a custom implementation (see [3.3.4 Implementing tipcConfigInfoGet\(\)](#), p.53). If you implement **tipcConfigInfoGet()**, you must add the file containing the implementation to your VxWorks Image Project.

Footprint Reduction and Building from Precompiled Libraries or from Source Code

If your installation includes the source code for TIPC, when you build Vxworks to include TIPC, you can build from precompiled TIPC libraries or from TIPC source code. You build from source code only if you are going to include one or more for the following build parameters for TIPC footprint reduction:

- **No TIPC debug** (INCLUDE_TIPC_NODEBUG)
- **No TIPC system messages** (INCLUDE_TIPC_NOSYS_MSGS)
- **No TIPC configuration service** (INCLUDE_TIPC_NOCFG_SERVICE)
- **No TIPC socket API** (INCLUDE_TIPC_NOSOCKET)

If you include **No TIPC socket API**, when you click **Finish** after selecting components ([Step 8](#)), Workbench will generate an error and you will need to explicitly exclude the **TIPC socket API** build component from your build.

Choose one of the following build options:

- **Build TIPC from object library** (default)

This option builds your VIP project using the TIPC library as originally installed. It does not apply if you include any of the footprint-reduction components listed earlier or if you include TIPC System Viewer instrumentation in your build.

- **Build TIPC from source**

This component is visible in Workbench only if your installation includes TIPC source code. It allows you to build the TIPC code from source code and is automatically included if any of the footprint-reduction components listed earlier is included. It is also automatically included if you include TIPC System Viewer instrumentation in your build.

Media Types

Communication over Ethernet is available to all BSPs. The availability of shared memory or distributed shared memory (DSHM) for TIPC communication depends on the BSP that you are building for. For information on media types, see [3.2.4 TIPC Media Types](#), p.32.

Include one or more of the following:

- **Ethernet** (default)
- **Shared Memory**
- **DSHM primary interface**

Network Stack With or Without UDP/IP and TCP/IP Services

You can choose include either the full VxWorks network stack in your build or, if you do not need either UDP/IP or TCP/IP services and you want a smaller VxWorks image, you can include a minimal stack with only the services required by TIPC.

Choose one of the following options:

- **TIPC and IP network stacks present** (default)
- **TIPC network stack only**

TIPC network stack only reduces the size of the network stack by approximately 380 KB. Including **TIPC network stack only** requires that you have previously excluded the **Network Components** bundle from your build (see [Step 4](#)). If you get the following error, it means that the network components were not excluded:

incompatible with Boot parameter process (INCLUDE_NET_BOOT)

For information on minimizing TIPC's footprint and the overall image size of your project, see [3.2.1 TIPC Footprint Reduction](#), p.27.

7. Include additional components from the following list (see, also, [Figure 3-3](#)) based on TIPC requirements and the needs of your project.
 - **TIPC configuration and display routines**
Enables the use of the **tipcConfig** utility and TIPC command-line show routines (see [3.2.6 TIPC configuration and display routines Build Component](#), p.42).
 - **TIPC inventory simulation demo**
Adds the TIPC inventory simulation, a sample application (see [E. Sample TIPC Application](#)), to a VxWorks kernel image.
 - **TIPC memory pool** (default)
A required component that allows you to allocate buffer space dedicated to TIPC sockets at startup. For further information, see [3.2.3 TIPC memory pool Build Component](#), p.31.
 - **TIPC prioritized interfaces**
Available only with BSPs that are compatible HEND interfaces. See [3.2.8 TIPC prioritized interfaces Build Component](#), p.45.
 - **TIPC socket API** (default)
You must explicitly exclude this component if you include the **No TIPC socket API** build component (see [3.2.2 TIPC socket API Build Component](#), p.30).
 - **TIPC System Defines**
Allows you to set TIPC parameter values replacing initial system defaults (see [3.2.7 Setting TIPC System Values](#), p.42).
 - **TIPC test suite demo** (default)
Adds the TIPC test suite (see [8. Using the TIPC Test Suite](#)).
8. Click **Finish** to include your selections in the build.
9. If you want to include System Viewer instrumentation in your build, you need to include the **TIPC instrumentation** (**INCLUDE_WVTIPC**) build component in your build.

The path to the **TIPC instrumentation** component is:

development tool components > System Viewer components > TIPC instrumentation

For more information, see [7.4 Including TIPC System Viewer Instrumentation in a VxWorks Image Project](#), p.104

10. Enter parameter values for any of the following components included in your build:
 - **TIPC memory pool**
Specify the number and size of memory buffers allocated at startup for use with TIPC sockets. For information, see [3.2.3 TIPC memory pool Build Component](#), p.31.
 - **TIPC prioritized interfaces**
For the parameter **TIPC interfaces using H-END**, enter a string value that lists all interfaces that are to be prioritized. For more information, see [3.2.8 TIPC prioritized interfaces Build Component](#), p.45
 - **TIPC socket API**
Specify the maximum number of concurrent sockets supported on a node (**Number of TIPC sockets**) and the number of incoming messages that can be queued on TIPC sockets (**Socket Receive Queue Threshold**). For more information, see [3.2.2 TIPC socket API Build Component](#), p.30.
 - **TIPC system defines**
Set TIPC system values (see [3.2.7 Setting TIPC System Values](#), p.42).
 - **TIPC static configuration**
Enter a TIPC configuration string for setting TIPC parameter values (see [3.3.1 Setting Parameters in the TIPC Configuration String](#), p.48).
 - **DSHM primary interface**
Set configuration parameters for memory allocation and other aspects of DSHM (see [Communication Using Distributed Shared Memory \(DSHM\)](#), p.35).
 - **Shared Memory**
Set configuration parameters for memory allocation and other aspects of shared memory (see [Shared Memory Communication](#), p.33).
11. Select **Build All** from the **Project** menu to build your project.

4

Using tipcConfig to Configure and Monitor TIPC

[4.1 Introduction 63](#)

[4.2 tipcConfig Syntax and Command Options 64](#)

4.1 Introduction

The **tipcConfig** utility allows you to dynamically configure a number of TIPC features, display current configuration settings, and monitor the behavior of links and nodes in a TIPC network. To have access to the **tipcConfig** utility, you need to build VxWorks to include the **TIPC configuration and display routines** (**INCLUDE_TIPC_SHOW**) build component (see [Table 3-1](#) under [3.2 Wind River TIPC Build Components](#), p.20).

4.2 tipcConfig Syntax and Command Options

The syntax for using **tipcConfig** is:

```
tipcConfig "[-]command_option[=arguments][ [-]command_option[=arguments]]  
[ [-]command_option[=arguments]]..."
```

Note the following:

- The entire command following **tipcConfig** must be enclosed in quotes.
- Each command option and its arguments must be separated from the next command option and its arguments by either a space or by a semi-colon.
- A command option and its arguments are always linked by an equals sign, as in the following example:

```
tipcConfig "-v;-max_ports=5000;max_nodes=350;netid=11"
```

- A command option can be preceded by **-**, as in **-v**, but this is optional.
- Command options can be abbreviated as long as they are unambiguous. For example **-addr** can be abbreviated to **-a**, but an abbreviation of either **-bd** or **-be** to **-b** would be ambiguous.

The following are equivalent examples of a **tipcConfig** command:

```
tipcConfig "-v -mng=enable -nt=ports,10,5,15"  
tipcConfig "v mng=enable nt=ports,10,5,15"
```

[Table 4-1](#) lists the **tipcConfig** command options in alphabetical order. Note, however, that there are restrictions on the order in which some options can be entered (see [4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands](#), p.75).

Table 4-1 **tipcConfig** Command Options

tipcConfig Option	Arguments	Description
-a or -addr	[= <i>node_address</i>] <i>node_address</i> is given in <Z.C.N> format.	If <i>node_address</i> is given, sets the local node's address. If no address is specified, gets the local node's address. You must always set the -addr option before you set the -be option (see 4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands , p.75).
-b	N/A	Lists the bearer or bearers associated with the local node or (with the -dest option) with the destination node. Example: -> tipcConfig "b" Bearers: eth:fei0
-bd	= <i>bearer_name</i> [, <i>bearer_name</i> [, <i>bearer_name</i> [,...]]] <i>bearer_name</i> is given in one of the following formats: <ul style="list-style-type: none">▪ eth:<i>interface_name</i> (Ethernet)▪ sm:sm0 (shared memory)▪ dshm:plb0 (DSHM)	Disables use of the specified bearer or bearers. Example: "bd=eth:fei0,dshm:plb0"
-be	= <i>bearer_name</i> [/ <i>ldomain</i> [/ <i>priority</i>]]][, <i>bearer_name</i> [/ <i>ldomain</i> [/ <i>priority</i>]]][, <i>bearer_name</i> [/ <i>ldomain</i> [/ <i>priority</i>]]...] For information on the individual syntax components, see 4.2.2 The -be Command Option , p.75.	Enables use of the specified bearer or bearers. For more information, see 4.2.2 The -be Command Option , p.75. The -addr option must always be set before the -be option (see 4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands , p.75).

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-d or -dest	[= <i>destination_node_addr</i>] <i>destination_node_addr</i> is given in Z.C.N format, for example: -dest=1.1.7 .	Allows you to specify the address of a node for which remote management is enabled and then enter tipcConfig command options for managing the specified node. In the current release, remote management is limited to a subset of tipcConfig command options and only allows you to display information about the remote node. For more information, see 4.2.7 Remote Management , p.84. If no destination address is specified, the destination address currently in effect is displayed. If no destination address has been set, the local node's address is displayed.
-h or -help	N/A	Displays a tabular listing of command options, similar to this table. Use this command option alone, without other command options.
-i	N/A	Toggles interactive mode. If interactive mode is in effect, when you enter a value to set a parameter, you are asked to confirm the operation, as in the following example: <pre>-> tipcConfig "-i -mng=enable" enable remote management [Y/n]</pre> Interactive mode stays in effect across uses of tipcConfig until it is disabled.
-l	= <i>[node_addr]</i> <i>node_address</i> is given in <Z.C.N> format.	If a node address is given, displays links to the specified node from either the local node or (with the -dest option) the destination node.

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-log	[= <i>size</i>] <i>size</i> , in bytes; an integer from 0 to 32768. The default value is 0.	<p>If a log size is specified, sets the size of the log. A log size of 0 (the default) means that logging is turned off.</p> <p>If there is a log, the minimum size is 512 bytes. If you specify a log size less than the minimum (other than 0), the minimum size is used.</p> <p>If you enter a log size and later change the size, the log is reset to empty and the new size goes into effect.</p> <p>If you enter -log without specifying a size, the current contents of the log is displayed and the log is reset to empty. For an example of log output, see 4.2.5 Sample Log Output, p.83.</p>

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-lp	<p><i>=link_name priority</i></p> <p><i>link_name</i> is given as shown in the entry for -ls.</p> <p><i>priority</i> is a value from 0 to 32. The default value is 32. See the Description column for the way priority is applied.</p>	<p>Sets the specified link's priority, as follows:</p> <ul style="list-style-type: none"> ▪ For priorities 1 through 31, the higher the integer, the greater the priority. ▪ A priority of 32 (the default) sets priority equal to the default priority of the medium (Ethernet, shared memory, or DSHM). <ul style="list-style-type: none"> – The default value for Ethernet is 10. – The default value for shared memory is 15. – The default value for DSHM is 15. – You cannot assign the same priority to more than two interfaces on a node.
-ls	<p><i>=link_name</i></p> <p><i>link_name</i> is given as:</p> <p><i>node_addr:if_name-dest_node:bearer</i></p> <p>Example:</p> <p>1.1.7:fei0-1.1.58:eth0</p>	<p>Displays usage statistics for the specified link on the local node or (with the -dest option) the destination node.</p> <p>For an example of output, see 4.2.6 Sample Output for the "ls" (Link Statistics) Option, p.83.</p>
-lsr	<p><i>=link_name</i></p> <p><i>link_name</i> is given as shown in the entry for -ls.</p>	<p>Resets statistics counters to zero for the specified link.</p>

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-lt	<p>=<i>link_name/tolerance_interval</i></p> <p><i>link_name</i> is given as shown in the entry for -ls.</p> <p><i>tolerance_interval</i> is given in milliseconds. You can set a value in the range from 50 to 30000 ms. The default value is 1500 ms for both Ethernet and shared memory.</p>	<p>Sets link <i>tolerance</i>. Tolerance is the minimum length of time, in milliseconds, that the node will wait before declaring that a link is down, if no communication is received on it. Within the time interval, the node makes multiple attempts to elicit communication on the link.</p>
-lw	<p>=<i>link_name/window_size</i></p> <p><i>link_name</i> is given as shown in the entry for -ls.</p> <p><i>window_size</i> is an integer in the range from 16 to 150. Default window size is 50.</p>	<p>Sets the window size for the link. <i>window_size</i> is the number of messages sent on the specified link that the node will keep in memory without needing to receive an acknowledgement from the recipient.</p>
-m	N/A	<p>Lists the media—Ethernet, shared memory, or both—on the local node or (with the -dest option) on the destination node.</p>
-max_clusters	<p>[=<i>max_number</i>]</p> <p><i>max_number</i> is an integer from 1 to 4095. The default value is 8.</p>	<p>The maximum number of clusters in this node's zone. If no <i>max_number</i> is specified, displays the current value for <i>max_number</i>.</p> <p>-max_clusters must always precede -addr (see 4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands, p.75).</p>

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-max_nodes	[= <i>max_number</i>] <i>max_number</i> is an integer from 8 to 4095. The default value is 255.	<p>If <i>max_number</i> is specified, sets the maximum number of nodes in this node's cluster. Typically, each node in a cluster has the same max nodes setting. If no <i>max_number</i> is specified, displays the current value for <i>max_number</i>.</p> <p>-max_nodes must always precede -addr (see 4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands, p.75).</p>
-max_ports	[= <i>max_number</i>] <i>max_number</i> is an integer from 127 to 65535. The default value is 8191.	<p>If <i>max_number</i> is specified, sets the maximum number of ports that his node can create. The number should include both incoming and outgoing ports for services offered by the node and a small number of additional ports needed by TIPC for system purposes. The number of ports needed for system purposes can vary, but is generally less than ten.</p> <p>If no <i>max_number</i> is specified, displays the current value for <i>max_number</i>.</p> <p>The precedence order for -max_ports is: max_ports > netid > a > be</p>
-max_publ	[= <i>max_number</i>] <i>max_number</i> is an integer from 1 to 65535. The default value is 10000.	<p>If <i>max_number</i> is specified, sets the maximum number of services a node can offer at one time. This is equivalent to the maximum number of port names and name sequences that all applications on the node, combined, can publish.</p> <p>If no <i>max_number</i> is specified, displays the current value for <i>max_number</i>.</p>

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-max_remotes	[= <i>max_number</i>] <i>max_number</i> is an integer from 0 to 255. The default value is 8.	If <i>max_number</i> is specified, sets the maximum number of nodes that this node can have links to outside its own cluster. If no <i>max_number</i> is specified, displays the current value for <i>max_number</i> .
-max_subscr	[= <i>max_number</i>] <i>max_number</i> is an integer from 1 to 65535. The default value is 2000.	If <i>max_number</i> is specified, sets the maximum number of subscriptions the local node supports. This is equivalent to the maximum number of services that all applications on the node, combined, can subscribe to. For information on subscriptions, see 2.7 Subscriptions , p.18. If no <i>max_number</i> is specified, displays the current value for <i>max_number</i> .
-max_zones	[= <i>max_number</i>] <i>max_number</i> is an integer from 1 to 255. The default value is 4.	If <i>max_number</i> is specified, sets the maximum number of zones in this node's network. If no <i>max_number</i> is specified, displays the current value for <i>max_number</i> . -max_zones must always precede -addr (see 4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands , p.75).

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-mng	[=enable disable]	<p>Specifies whether remote management is enabled or disabled for this node. If no argument is entered, the current state is displayed, either enabled or disabled.</p> <p>When remote management of a node is enabled, other nodes in the network can use the tipcConfig utility to manage the node and display information about it. In the current release, remote management is limited to a subset of tipcConfig command options and only allows you to display information about a managed node. For more information, see 4.2.7 Remote Management, p.84.</p> <p>Note that remote management is enabled by default, unless you reset the default through the Remote Management (TIPC_DEF_REMOTE_MGT) parameter (see the table entry for Remote Management, p.44).</p>
-n	[=lookup_domain] lookup_domain is given in <Z.C.N> format.	<p>If <i>lookup_domain</i> is given, lists all nodes known to the current node within the specified domain. For information about lookup domains, see 2.5.4 Address Resolution, p.15.</p> <p>If no lookup domain is given, displays all known nodes within the network.</p>

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-netid	[= <i>network_id</i>]	<p>If <i>network_id</i> is specified, sets the network address of the node.</p> <p>For a usage example in which separate networks within a LAN are set up, see 4.2.8 Using the -netid Option to Set Up Separate TIPC Networks Within a LAN, p.86.</p> <p>If no network ID is specified, gets the network address.</p> <p>The default network ID at initial startup is 4711.</p>
-nt	<p>[=<i>type</i>[,<i>low</i>[,<i>up</i>]]] <i>depth</i>[,<i>type</i>[,<i>low</i>[,<i>up</i>]]]</p> <p>where <i>depth</i> is one of the following:</p> <ul style="list-style-type: none"> ▪ types ▪ names ▪ ports ▪ all 	<p>Lists information in the name table of the local node or (with the -dest option) the name table of the destination node. For further information and sample output, see 4.2.9 The -nt Command Option, p.86.</p>
-p	N/A	<p>For each port, the associated port name is given, if there is one. If a port is currently connecting to another port, the port ID of the connecting port is given.</p> <p>For sample output, see 4.2.10 Sample Output for the -p (Ports) Option, p.89.</p>

Table 4-1 **tipcConfig Command Options** (cont'd)

tipcConfig Option	Arguments	Description
-s	N/A	<p>Displays the current TIPC release number.</p> <p>Examples:</p> <pre>-> tipcConfig "-s" TIPC version 1.7.5</pre> <pre>-> tipcConfig "-dest=1.3.1;-s" Status for node <1.3.1>: TIPC version 1.7.5</pre>
-v	N/A	<p>Toggles verbose mode. In verbose mode, the system displays a confirmation for new settings, as in the following example:</p> <pre>-> tipcConfig "v" verbose mode: active</pre> <p>Verbose mode stays in effect across uses of tipcConfig until it is disabled.</p>
-V	N/A	<p>Displays the current version of the tipcConfig utility.</p> <p>Example:</p> <pre>> tipcConfig "-V" TIPC configuration tool version 1.1.4</pre>

4.2.1 Constraints on the Ordering of Command Options in **tipcConfig** Commands

There are constraints on the order in which some **tipcConfig** command options can be specified. These constraints apply to the options within a single **tipcConfig** command and also to the use of options across **tipcConfig** commands. For example, the **-addr** option must be set before the **-be** (bearer) option, as in the following **tipcConfig** command:

```
max_nodes=100;max_ports=200;netid=1000;a=1.1.27;log=1024;be=eth:cpm0
```

The following sequence of **tipcConfig** commands is also valid:

```
max_ports=200;netid=1000;a=1.1.27;
max_nodes=100;log=1024;be=eth:cpm0
```

The rules for entering **tipcConfig** command options are:

- If entered, the **-v** (verbose), **-i** (interactive), and **-dest** options should be specified before other command options.
- Within a single **tipcConfig** command and across multiple **tipcConfig** commands, the following precedence relations must be observed:

```
max_ports > netid > a > be
```

```
max_zones, max_nodes, max_clusters > a
```

where “>” means “precedes”.

4.2.2 The **-be** Command Option

The **tipcConfig -be** option enables TIPC communication over one or more interfaces using Ethernet, shared memory, or distributed shared memory.

The syntax of the **-be** option with its arguments is:

```
-be=bearer_name[/domain[/priority]][,bearer_name[/domain[/priority]]
[,bearer_name[/domain[/priority]]...]
```

where:

The **tipcConfig -be** option enables TIPC communication over one or more interfaces using Ethernet, shared memory, or distributed shared memory. An individual interface can be used for only one media type (see [3.2.4 TIPC Media Types](#), p.32), but you can configure separate interfaces on a node for different media types (see [3.2.4 TIPC Media Types](#), p.32). You can also configure two or more

interfaces for the same media type, excluding DSHM, which does not support multiple links (see [2.2.3 Multiple Links for Load-Sharing and Switchover](#), p.8).

The syntax of the **-be** option with its arguments is:

```
-be=bearer_name[/domain[/priority]][,bearer_name[/domain[/priority]]  
[.bearer_name[/domain[/priority]]...]]
```

where:

- *bearer_name* is the media type and interface or bus name, as follows:
 - For Ethernet, enter **eth:interface_name**. For example: **-be=eth:eth1**
 - For shared memory, always enter: **-be=sm:sm0**.
 - In the current release, for distributed shared memory, always enter: **-be=dshm:plb0**.

Note that shared memory and DSHM are not available with all BSPs (see [3.2.4 TIPC Media Types](#), p.32).

- *domain* is an optional argument given in <Z.C.N> format that determines which nodes the current node can have links to.

In specifying a domain, a zero value for Z, C, or N, means that the domain includes all zones, clusters, or nodes:

- If *domain* is specified as <0.0.0>, all nodes in the network are included in the domain.
- If *domain* is specified as <1.0.0>, the domain is restricted to zone 1, but includes all nodes in all clusters within zone 1.
- If *domain* is specified as <Z.C.0>, where Z and C are the current node's zone and cluster, the domain includes only the nodes within the current node's cluster.

If no domain is specified, the current node can only have links to other nodes in its cluster. This is equivalent to a domain of Z.C.0, where Z is the node's own zone and C is the node's own cluster.

For more detailed information on domains, see [4.2.3 Specifying a Domain](#), p.77.

- *priority* assigns a priority to communication over the specified interface, as follows:
 - For priorities 0 through 31, the higher the integer, the greater the priority.
 - A priority of 32 (the default) sets priority equal to the default priority of the medium (Ethernet or shared memory).

- The default value for Ethernet is 10.
- The default value for shared memory is 15.
- The default value for DSHM is 15.

If no priority is specified, the default priority is 0.



NOTE: You cannot assign the same priority to more than two interfaces on a node.

Before setting the **-be** option, you must always set the **-addr** option (see [4.2.1 Constraints on the Ordering of Command Options in *tipcConfig* Commands](#), p.75).

4.2.3 Specifying a Domain

When TIPC communication over a bearer is enabled, TIPC broadcasts messages over the bearer for the purpose of detecting other nodes in the network that use the same medium and establishing links to them. TIPC establishes links between nodes based on the *domains* assigned to their bearers.

When you configure a bearer by setting the **be** parameter in a configuration string [3.3.2 Setting the *be* \(bearer\) Parameter](#), p.51 or using the **-be** command option with the **tipcConfig** utility, you can specify a *domain* for the bearer. The domain is entered in *Z.C.N* format and determines the nodes that bearer's own node can have links to:

- If two nodes have bearers that use the same medium (for example, Ethernet) and the domain of each bearer includes the network address of the other node, TIPC establishes a link between the nodes.

In specifying a domain, a zero value for *Z*, *C*, or *N*, means that the domain includes all zones, clusters, or nodes:

- If *domain* is specified as <0.0.0>, all nodes in the network are included in the domain.
- If *domain* is specified as <1.0.0>, the domain is restricted to zone 1, but includes all nodes in all clusters within zone 1.
- If *domain* is specified as <*Z*.*C*.0>, where *Z* and *C* are the current node's zone and cluster, the domain includes only the nodes within the current node's cluster.

If no domain is specified, the current node can only have links to other nodes in its cluster. This is equivalent to a domain of *Z.C.0*, where *Z* is the node's own zone and *C* is the node's own cluster.

Domain Settings for Meeting TIPC Network Requirements

TIPC requires that all the nodes in a cluster have direct links to each other, that all the clusters in a zone have direct links to each other, and that all the zones in a network have direct links to each other.

Domain Settings for Direct Links Between Nodes within a Cluster

The simplest way to ensure that all nodes in a cluster have direct links to each other over a given medium is to not specify any domain and simply accept the default setting. However, this also means accepting the default priority for a medium, which may not be desirable.

To assign the default domain and a priority to a bearer, you need to enter the default domain as *Z.C.0*, where *Z* is the node's own zone and *C* is the node's own cluster. For example, the following setting configures an Ethernet bearer on node <1.1.5> with the default domain and a priority of 12:

```
be=eth:eth0/1.1.0/12
```

It is also possible to assign a bearer a domain that allows linkage only to a specific node within the bearer's cluster, as in the following example for node <1.1.5>:

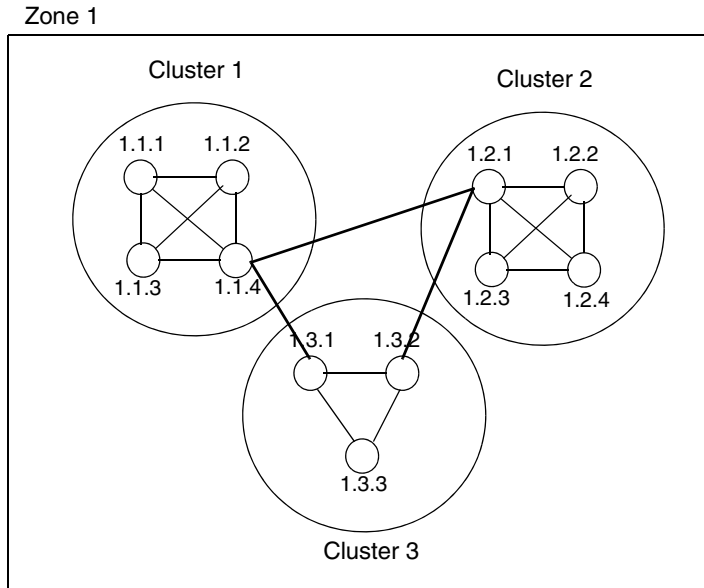
```
be=eth:eth1/1.1.4
```

This can be useful if, for example, you have multiple interfaces on a node. You can configure one interface as an Ethernet bearer and assign it the default domain. This provides for links to all nodes in its cluster. You can configure a second interface as an Ethernet bearer and assign it a restricted domain that creates a link to one specific node for purposes of load-sharing and switchover.

Domain Settings for Links Between Clusters

In a zone with multiple clusters, each cluster must have direct links to all other clusters. [Figure 4-1](#) illustrates two ways of assigning such links.

Figure 4-1 **Links Between Clusters**



In clusters 1 and 2, a single node acts as a router to multiple clusters. This is likely to be the most common way of handling routing between clusters. The alternative is to have separate nodes act as routers to different clusters, as in cluster 3.

To have a single node act as a router to multiple clusters, enter the domain as follows:

Z.0.0

where Z is the zone of the routing node.

To have a node act as a router to a single cluster in its own zone, enter the domain as follows:

Z.C.N

where:

- Z is the zone of the routing node.
- C is the external cluster.
- N is a routing node in the external cluster.

The following table shows the domains that would need to be assigned to bearers for each of the routing nodes in [Figure 4-1](#).

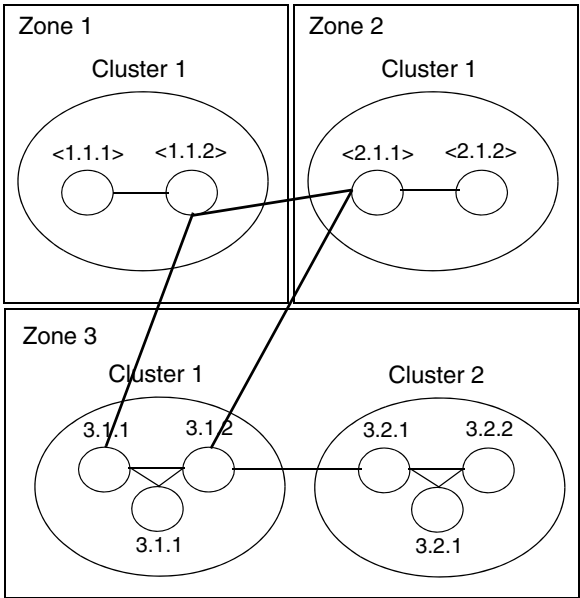
Table 4-2 **Cross-Cluster Domain Assignments**

Node	Domain
1.1.4	1.0.0
1.2.1	1.0.0
1.3.1	1.1.4
1.3.2	1.2.1

Domain Settings for Links Between Zones

In a network with multiple zones, each zone must have direct links to all other zones. [Figure 4-2](#) illustrates two ways of assigning such links.

Figure 4-2 **Links Between Zones**



In zones 1 and 2, a single node acts as a router to multiple zones. This is likely to be the most common way of handling routing between zones. The alternative is to have separate nodes act as routers to different zones, as in zone 3.

To have a single node act as a router to multiple zones, enter the domain as follows:

0.0.0

To have a node act as a router to a single external zone, enter the domain as follows:

Z.C.N

where:

- Z is the external zone.
- C is a cluster in the external zone.
- N is a routing node in the external zone.

The following table shows the domains that would need to be assigned to bearers for each of the routing nodes in [Figure 4-2](#).

Table 4-3 **Cross-Zone Domain Assignments**

Node	Domain
1.1.2	0.0.0
2.1.1	0.0.0
3.1.1	1.1.2
3.1.2	2.1.1

4.2.4 The -dest Command Option

If remote management has been enabled on another node (see the table entry for [-mng](#), p.72), this option allows you to enter the node's address in <Z.C.N> format and enter command options for managing the designated node. In the current release, remote management is limited to a subset of **tipcConfig** command options and only allows you to display information about the remote node. The command options available for remote management are:

The syntax of the **-dest (-d)** command option is:

-dest[=*destination_node_addr*]

where *destination_node_addr* is given in <Z.C.N> format.

In the current release, the remote node is limited to the following **tipcConfig** commands:

Table 4-4 **Command Options for -dest**

-b (get bearers)
-l (get links to clusters)
-ls (get link statistics)
-m (get media)
-n (get nodes in cluster)
-nt (show name table)
-p (get port information)
-s (display TIPC release number)

When you specify the address of a remote node, **tipcConfig** command options apply to the specified node. In addition, all subsequent **tipcConfig** commands also apply to the remote node, until you reset the **-dest** option to the local node.

To reset the **-dest** option to the local node do one of the following:

- Enter **tipcConfig -dest** with the destination address set to the address of the local node.
- Enter **tipcConfig -dest** with the destination address set to <0.0.0>, as in the following example:

tipcConfig (" -dest=0.0.0;nt=ports,10,5,15")

If you use the **-dest** option without specifying a destination address, the destination address currently in effect is displayed. If no destination address has been set, the local node's address is displayed.

4.2.5 Sample Log Output

The following is sample output for the **tipcConfig "log"** command:

```
-> tipcConfig "log"
Log dump:
TIPC info: Established link <1.1.7:fei0-1.1.58:eth0> on network plane A
TIPC info: Lost link <1.1.7:fei0-1.1.58:eth0> on network plane A
TIPC info: Lost contact with <1.1.58>
TIPC info: Disabled bearer <eth:fei0>
TIPC info: Own node address <1.1.7>, network identity 1960
TIPC info: Enabled bearer <eth:fei0>, discovery domain <1.1.0>
```

In the log:

- **network plane A** is a TIPC-assigned label for the bearer; each bearer gets a similar label.
- A discovery domain is effectively the same as a lookup domain. For information about lookup domains, see [2.5.4 Address Resolution](#), p.15.

For a list of the messages that can occur in the log, see [F. TIPC Log Messages](#)

4.2.6 Sample Output for the “ls” (Link Statistics) Option

The following is sample output for the **tipcConfig "ls"** command:

```
-> tipcConfig "ls=1.1.7:fei0-1.1.58:eth0"
Link <1.1.7:fei0-1.1.58:eth0>
  ACTIVE MTU:1500 Priority:10 Tolerance:1500 ms Window:50 packets
  RX packets:2 fragments:0/0 bundles:0/0
  TX packets:79 fragments:0/0 bundles:0/0
  TX profile sample:82 packets average:60 octets
  0-64:100% -256:0% -1024:0% -4096:0% -16354:0% -32768:0% -66000:0%
  RX states:1840 probes:927 naks:0 defs:0 dups:0
  TX states:1840 probes:913 naks:0 acks:0 dups:0
  Congestion bearer:0 link:0 Send queue max:3 avg:0
```

In the output:

- For the **Tolerance** and **Window** values, see the entries for **-lt** and **-lw** in [Table 4-1](#).
- For received messages (**RX states**):
 - **probes** are messages sent to check on whether a link is still valid, when there has been no traffic or response from another node.
 - **defs** are the number of messages received out of order. They are deferred packets that are held in queue until all missing packets are received.

- **dups** are the number of duplicate messages received.
- For transmitted messages (**TX states**):
 - **probes** are messages sent to check on whether a link is still valid, when there has been no traffic or response from another node.
 - **dups** are the number of messages retransmitted.

4.2.7 Remote Management

When remote management of a node is enabled through the **tipcConfig -mng** command option (see [Enabling Remote Management](#), p.84), other nodes in the network can use the **tipcConfig -dest** command option to initiate management of the node and to display information about it (see [The -dest Command Option for Specifying the Address of a Node to be Managed](#), p.85). In the current release, remote management is limited to a subset of **tipcConfig** command options and only allows you to get information about a remote node, not to alter its configuration.

tipcConfig Command Options available for remote management

The following **tipcConfig** command options are available for remote management in the current release:

- b (get bearers)
- l (get links to clusters)
- ls (get link statistics)
- m (get media)
- n (get nodes in cluster)
- nt (show name table)
- p (get port information)
- s (display TIPC release number)

Enabling Remote Management

When you build VxWorks to include TIPC, remote management of TIPC nodes is enabled by default. You can change this and build VxWorks with TIPC remote management disabled by default through the **Remote Management** (TIPC_DEF_REMOTE_MGT) build parameter (see the table entry for [Remote Management](#), p.44). At any time after startup, you can use the **tipcConfig -mng**

command option to disable or enable remote management of an individual node (see the table entry for *-mng*, p.72).

The **-dest** Command Option for Specifying the Address of a Node to be Managed

To manage a node for which remote management has been enabled, you use the **tipcConfig -dest** (or **-d**) command option from a remote node and specify the network address of the node to be managed in Z.C.N format. For example:

-dest=1.1.25

Enter the **-dest** command option before any other command options, except for the **-v** and **-i** command options, which can come before it (see [4.2.1 Constraints on the Ordering of Command Options in tipcConfig Commands](#), p.75).

The following example specifies node <1.2.5> as the destination node for remote management and displays information about the node's bearers and links:

tipcConfig "-dest=1.2.5;-b;-l"

Once you use the **-dest** command option to specify a destination node, all subsequent uses of the **tipcConfig** utility apply to the specified node, unless you reset the destination. To reset the destination, you need to use the **tipcConfig** command with the **-dest** command option and specify a new destination for **tipcConfig** commands. To reset the destination to the local node, you can enter either the local node's address or, simply, **0.0.0** as the address. For example if the local node's address is <1.1.7>, the following are equivalent commands to reset the destination to the local node and display information about its bearers and links:

tipcConfig "-dest=1.1.7;-b;-l"

tipcConfig "-dest=0.0.0;-b;-l"

Note that if the local node is <1.1.7> the following command to first get information about node <1.3.17> and then reset management to the local node is not valid: **tipcConfig "-dest=1.3.17;-b;-l;-dest=1.1.7"**. You need to use separate **tipcConfig** commands with separate command strings specifying the **-dest** command option.

4.2.8 Using the **-netid** Option to Set Up Separate TIPC Networks Within a LAN

You can use the **tipcConfig -netid** option to set up independent TIPC networks within a LAN of interconnected TIPC nodes. This can be useful for development purposes.

For example, suppose you have a LAN with five TIPC nodes and want to work with only three of them for development purposes. To do this, you use the **-netid** option to assign each node the same network ID and you use the **-addr** option to assign each node a network address within the network. The following example assigns a network ID and a network address to a node:

```
tipcConfig "-netid=1000 -addr=1.1.7"
```

4.2.9 The **-nt** Command Option

The **tipcConfig -nt** option lists information in the name table of the local node or (with the **-dest** option) in the name table of the destination node. For sample output see [Sample Output for the -nt Command Option](#), p.87. The syntax for specifying arguments is:

```
tipcConfig -nt[=type[,low[,up]]] | depth[,type[,low[,up]]]
```

where

- *depth* is one of the following:
 - **types**
List only the port types in the node's name table.
 - **names**
List only the port names (consisting of both a type and an instance) in the node's name table.
 - **ports**
List port names and their port identities.
 - **all**
List port names, port identities, and publication information.

If no *depth* is specified, the output is the same as for a *depth* of **all**.

- *type* is a specific type for which you want to display information.

- *low* and *up* are optional lower and upper instance values for a port name sequence for which you want to display information. To restrict the display of information to a single port name instance, you can enter a single value. For example, the following displays name table information for a port whose type is 5 and instance value is 8:

```
tipcConfig "nt=5,8"
```

If you do not specify an argument, **tipcConfig -nt** lists all the information in the name table. For sample output, see [Sample Output for the -nt Option Without Arguments \(Equivalent to "-nt=all"](#), p.88.

Sample Output for the -nt Command Option

The following sections give examples of output from several uses of the **-nt** command option.

Sample Output for depth=types

The following is sample output for **tipcConfig -nt** with a depth of **types**:

```
-> tipcConfig ("-nt=types")
Type
-----
0
1
```

Sample Output for depth=names

The following is sample output for **tipcConfig -nt** with a depth of **names**:

```
-> tipcConfig ("-nt=names")
Type      Lower      Upper
-----
0          16781319    16781319
           16781322    16781322
1          1          1
```



NOTE: As described in [2.7 Subscriptions](#), p.18, type 0 is a special type whose instances correspond to nodes within the network. In the output for the **-nt** option, the instance values for type 0 are decimal representations of a node's <Z.C.N> address, where Z is 8 bits, C is 12 bits, and N is 12 bits.

Sample Output for depth=ports

The following is sample output for **tipcConfig -nt** with a depth of **ports**:

```
-> tipcConfig ("-nt=ports")
```

Type	Lower	Upper	Port Identity
0	16781319	16781319	<1.1.7:1086734332>
	16781322	16781322	<1.1.10:1086734332>
1	1	1	<1.1.10:1086734333>

For the interpretation of type 0 instance values, see the note in [Sample Output for depth=names](#), p.87.

Sample Output for the -nt Option Without Arguments (Equivalent to “-nt=all”

The following is sample output for **tipcConfig -nt** without any arguments, which is equivalent to using **-nt** with a depth of **all**:

```
-> tipcConfig "nt"
```

Type	Lower	Upper	Port Identity	Publication
0	16781319	16781319	<1.1.7:1086734332>	1086734333 zone
	16781322	16781322	<1.1.58:4038885369>	4038885370
1	1	1	<1.1.7:1086734333>	1086734334 node
75	0	0	<1.1.7:1086734027>	1086734028 cluster
	4	4	<1.1.7:1086734165>	1086734166 zone
			<1.1.7:1086734099>	1086734100 zone
	5	5	<1.1.7:1086734031>	1086734032 zone
	7	7	<1.1.7:1086734090>	1086734091 zone
		7	<1.1.7:1086734049>	1086734050 zone

The values in the publication column, a reference number and scope value, are for Wind River internal debugging.

For the interpretation of type 0 instance values, see the note in [Sample Output for depth=names](#), p.87.

Sample Output for the -nt Option with arguments for type, low, and up

The following example shows output for **tipcConfig -nt** when a port name sequence is specified using the **type**, **low**, and **up** arguments:

```
-> tipcConfig "nt=75,0,7"
```

75	0	0	<1.1.7:1086734027>	1086734028 cluster
	4	4	<1.1.7:1086734165>	1086734166 zone
			<1.1.7:1086734099>	1086734100 zone
	5	5	<1.1.7:1086734031>	1086734032 zone
	7	7	<1.1.7:1086734090>	1086734091 zone
			<1.1.7:1086734049>	1086734050 zone

For the interpretation of type 0 instance values, see the note in [Sample Output for depth=names](#), p.87.

4.2.10 Sample Output for the -p (Ports) Option

The **tipcConfig -p** option lists all ports on the local node or (with the **-dest** option) all ports on the destination node. The following is sample output:

```
-> tipcConfig "p"

Ports:
1086734333: bound to {1,1}
1086734332: bound to {0,16781319}
1086734319: connected to <1.1.7:1086734317> via {1,1}
1086734317: connected to <1.1.7:1086734319>
1086734165: bound to {75,4}
1086734150: bound to {75,7}
1086734099: bound to {75,4}
1086734048: connected to <1.1.7:1086734046> via {1,1}
1086734046: connected to <1.1.7:1086734048>
1086734040: bound to {75,8}
```

In the example, the values in parentheses are the type and instance values making up a port name. If a port has a current connection, the port it is connected to is given in angled brackets. If a connection does not show a port name, it means that the connection is made through a physical address and not through a functional address.

5

Subscriptions

5.1 Introduction 91

5.2 Creating and Using the TIPC Subscription Service 92

5.1 Introduction

When an application uses the **bind()** routine to bind a service to a socket, it assigns the service a functional address and a scope (see [2.5.3 Functional Addressing](#), p.13). All nodes within the scope of a service are notified when the service becomes available and keep track of its ongoing availability. Every TIPC node maintains a connection-oriented, message-based subscription service for providing information on the availability of the services it knows about. An application can subscribe to this service in order to learn about the availability or unavailability of individual services known to a node.

In the current release, there are two types of subscriptions:

- Port level (**TIPC_SUB_PORTS**)

When an application creates a port-level subscription for a service, it is notified each time the service becomes available or unavailable at a socket anywhere in the network, as long as the application's node is within the scope of the service.

- Service level (**TIPC_SUB_SERVICE**)

When an application creates a service-level subscription, it only receives a notification

- when a previously available service becomes unavailable throughout the network
- when a service previously unavailable throughout the network becomes available

An application can also use subscriptions to gather information about the topology of a TIPC network. TIPC automatically assigns every node in the network a port name in which the *type* is 0 and the *instance* value is the node's network address (<Z.C.N>) and publishes information about the availability of this port name, just as it does for other port names. By subscribing to port names of *type* 0, an application can receive notifications whenever a node joins or leaves the network.

5.2 Creating and Using the TIPC Subscription Service

This section describes how to create a subscription and how to receive and interpret an event notification when a service becomes available or unavailable. Once an application has received the subscription information it requires, it can terminate a subscription by calling **close()** to end the connection to the subscription service.

The definitions of structures and macros mentioned in this section can be found in [D. Header File Definitions](#).

5.2.1 Creating a Subscription

To create a subscription, an application:

- Creates a **SOCK_SEQPACKET** connection to the subscription service, using the reserved port name 1,1 (type 1, instance 1) as the destination address.
- Sends a message containing a **tipc_subscr** structure to the subscription service.

The two operations can be performed sequentially using the **connect()** routine followed by call to the **send()** routine. You can also use TIPC's implied

connection-handshake capability to combine the two operations in a single **sendto()** or **sendmsg()** operation (see **sendto()** and **sendmsg()** in [B. Socket and Utility Routines](#)).

The contents of the message sent to the subscription service must be a **tipc_subscr** structure. The structure is defined as follows:

```
struct tipc_name_seq seq;           /* name sequence of interest */
__u32 timeout;                     /* subscription duration (in ms) */
__u32 filter;                      /* bitmask of filter options */
char usr_handle[8];               /* available for subscriber use */
```

Instead of entering a duration in milliseconds for the **timeout** field, you can create a subscription with no time limit by specifying a timeout of **TIPC_WAIT_FOREVER**.

The **filter** field of the structure determines the type of subscription created. It currently supports two options, corresponding to service-level and port-level subscriptions (see [5.1 Introduction](#), p.91):

- **TIPC_SUB_PORTS**
- **TIPC_SUB_SERVICE**

5.2.2 Receiving a Subscription Event Notification

To receive an event notification when a service becomes available or unavailable, the application calls the **recv()** or **recvmsg()** routine. When a subscription event occurs, the application receives a **tipc_event** structure. The structure is defined as follows:

```
struct tipc_event {
    __u32 event;                     /* event type */
    __u32 found_lower;               /* matching name seq instances */
    __u32 found_upper;              /* " " " " " */
    struct tipc_portid port;         /* associated port */
    struct tipc_subscr s;            /* associated subscription */
}
```

The **event** field of the structure supports three types of subscription events:

- **TIPC_PUBLISHED** – the specified service is now available.
- **TIPC_WITHDRAWN** – the specified service is no longer available.
- **TIPC_SUBSCR_TIMEOUT** – the subscription has expired.

6

Using the Wind River VxWorks Simulator with TIPC

- 6.1 Introduction 95
- 6.2 Simulating a Standalone TIPC Node 96
- 6.3 Simulating a Network of TIPC Nodes 96

6.1 Introduction

The Wind River VxWorks Simulator allows you to simulate a single hardware target or a network with multiple targets for testing a VxWorks application. For TIPC, you can use a single instance of VxWorks simulator to simulate a standalone TIPC node or you can use multiple instances to simulate a network of TIPC nodes connected by Ethernet or shared memory.

6.2 Simulating a Standalone TIPC Node

To use VxWorks simulator to simulate a standalone TIPC node:

1. Build a VxWorks image containing TIPC (see [3. Building VxWorks to Include Wind River TIPC](#)).

You can use the TIPC default configuration without changes or you can modify it to suit your requirements. By default, the simulated node is assigned a default network address of <0.0.0>.

2. Start the simulated standalone target from a VxWorks development shell. For example:

```
> vxsim
```

6.3 Simulating a Network of TIPC Nodes

When you simulate a network of TIPC nodes, each node must have a unique TIPC network address and each node must have the same **be** (bearer) setting.

You can build TIPC to assign network addresses and a bearer either through TIPC static configuration or TIPC dynamic configuration (see [3.3 Configuring Wind River TIPC](#), p.46).

If you use Workbench, the simplest way to configure a simulated network of TIPC nodes is to build TIPC with dynamic **bootline configuration**. A special feature of bootline configuration is that when you use it with VxWorks simulator, it provides automatic configuration of network addresses and bearers (see [TIPC with bootline configuration, using default configuration values](#), p.97). If you build TIPC for static configuration only, you need to build a separate image for each TIPC node you want to simulate.

6.3.1 Simulating a Network of TIPC Nodes That Use Ethernet

To use VxWorks simulator to simulate a network of TIPC nodes that communicate over Ethernet:

1. Build a VxWorks image containing TIPC (see [3. Building VxWorks to Include Wind River TIPC](#)).

For use with VxWorks simulator, it is simplest if you build a VxWorks image with TIPC and **bootline configuration**.

You can include the **TIPC show routines** component in your build.

2. Configure a simulated Ethernet network. (For information on configuring a simulated Ethernet network, see *Wind River VxWorks Simulator User's Guide:Tutorials:Basic Simulated Network with Multiple Simulators*.)
3. Start the VxWorks simulator network daemon from the VxWorks development shell. For example:

```
> vxsimnetd
```

4. From a VxWorks development shell, use the **vxsim** command to start your simulated target nodes. The way you enter arguments for **vxsim** depends on whether you are using Wind River TIPC with bootline configuration and whether you want to use default values for TIPC configuration.

In all cases, you must enter **vxsim** with **-ni** and **-p** options, as follows:

- Specify the **-ni** option followed by the IP address of the simulated network device used by the target.
- Specify the **-p** option followed by a processor number for the target.

TIPC with bootline configuration, using default configuration values

If your VxWorks image includes TIPC bootline configuration, you can omit the **vxsim -o** option and **vxsim** will automatically provide a unique network address for each target node and set the bearer to **simnet0**.

The network address that **vxsim** assigns to a target node is always equivalent to:

$$a=1.1.p+1$$

where *p* is the processor number assigned with the **-p** option.

The following example assumes bootline TIPC configuration and shows **vxsim** commands for starting two simulated nodes with only the **-ni**, and **-p** options:

```
> vxsim -ni simnet=192.168.200.4 -p 4
```

```
> vxsim -ni simnet=192.168.200.5 -p 5
```

The first target in the example is assigned a processor number of 4, which means that it is assigned a network address of <1.1.5>. The second target is assigned a processor number of 5 and a network address of <1.1.6>.

TIPC with bootline configuration and manually entered configuration values

If you want to manually enter configuration values for any of the TIPC configuration parameters, you need to use the **vxsim -o** option followed by a TIPC configuration string (see [3.3.1 Setting Parameters in the TIPC Configuration String](#), p.48) containing a network address (**a=1.1.N**) and bearer (**be=eth:simnet0**) for the target node.

The following example shows **vxsim** commands for starting two simulated nodes with the **-ni**, **-p**, and **-o** options:

```
> vxsim -ni simnet=192.168.200.4 -p 4  
    -o "a=1.1.4;be=eth:simnet0"  
  
> vxsim -ni simnet=192.168.200.5 -p 5  
    -o "a=1.1.5;be=eth:simnet0"
```

TIPC configuration through tipcConfigInfoGet()

If you do not use TIPC with bootline configuration, configuration information is provided through the user-implemented **tipcConfigInfoGet()** routine (see [3.3.4 Implementing tipcConfigInfoGet\(\)](#), p.53). In this case, you need to specify the **vxsim -ni** and **-p** options, but you do not need to specify the **-o** option.

The following example assumes configuration through **tipcConfigInfoGet()** and shows **vxsim** commands for starting two simulated nodes with the **-ni**, and **-p** options:

```
> vxsim -ni simnet=192.168.200.4 -p 4  
  
> vxsim -ni simnet=192.168.200.5 -p 5
```

6.3.2 Simulating a Network of TIPC Nodes That Use Shared Memory

To simulate a network of TIPC nodes that communicate with each other using shared memory, you follow the procedure for simulating a network that uses Ethernet for communication, with the following differences:

- If you manually enter configuration values through the **vxsim -o** option (see [TIPC with bootline configuration and manually entered configuration values](#), p.98, in the preceding section), the bearer parameter must always be set as follows:
be=sm:sm0
- The **vxsim -p** option must be set sequentially from node to node, starting with 0 for the master node.

The following example shows **vxsim** commands for starting two simulated nodes that communicate with each other using shared memory:

```
> vxsim -p 0 -o "a=1.1.4;be=sm:sm0"  
> vxsim -p 1 -o "a=1.1.5;be=sm:sm0"
```

6.3.3 Simulating a Network of TIPC Nodes That Use DSHM

6

To simulate a network of TIPC nodes that communicate with each other using DSHM, follow the procedure for simulating a network that uses Ethernet for communication, with the following differences:

- If you manually enter configuration values through the **vxsim -o** option (see [TIPC with bootline configuration and manually entered configuration values](#), p.98, in the preceding section), the bearer parameter must always be set as follows:

be=dshm:plb0

- The **vxsim -p** option must be set sequentially from node to node, starting with 0 for the master node.

The following example shows **vxsim** commands for starting two simulated nodes that communicate with each other using shared memory:

```
> vxsim -p 0 -o "a=1.1.4;be=dshm:plb0"  
> vxsim -p 1 -o "a=1.1.5;be=dshm:plb0"
```


7

Using Wind River System Viewer with TIPC

- 7.1 Introduction 101
- 7.2 TIPC Events Covered by System Viewer 102
- 7.3 Event Levels 102
- 7.4 Including TIPC System Viewer Instrumentation in a VxWorks Image Project 104

7.1 Introduction

Wind River System Viewer is a logic analyzer for embedded software that makes it possible to visualize and troubleshoot complex target activities. If your VxWorks installation includes TIPC source code, you can use System Viewer to display and log TIPC socket events.

7.2 TIPC Events Covered by System Viewer

The TIPC events currently instrumented for System Viewer correspond to the following socket and I/O calls:

accept	listen	sendmsg
bind	read	sendto
close	recv	setsockopt
connect	recvfrom	shutdown
getsockopt	recvmsg	socket
ioctl	send	write

7.3 Event Levels

Within System Viewer, you specify an event level that determines the types of events displayed. The available event levels are **VERBOSE**, **INFO**, **WARNING**, **CRITICAL**, and **FATAL**. When you choose an event level, all events at or above the specified level are displayed. The following table lists event levels from lowest to highest and describes each level.

Table 7-1 **System-Viewer Event Levels**

Event Level	Description
VERBOSE	Events that occur frequently during normal operation, such as creating a socket or initializing a device. Such events, which provide the highest level of detail, often occur as a result of a user-level routine.
INFO	Events indicating minor checkpoints that occur during normal operation, such as the closure of a socket.
WARNING	Events indicating unusual situations that might cause later errors. Many events at this level occur in response to invalid or unexpected input from a remote host. The ability of TIPC processing to send to and receive data from other addresses is generally not affected.

Table 7-1 System-Viewer Event Levels (cont'd)

Event Level	Description
CRITICAL	Events indicating uncorrectable transient errors. Some events at this level can be avoided by altering the system configuration, but most cannot be prevented by any user action. Operations in process when such an event occurs will fail, and any TIPC data is usually discarded, but the event has no permanent effect on TIPC processing. Inability to allocate space for a new socket structure is a typical event at this priority level.
FATAL	Events resulting from unrecoverable errors which prevent completion of the current operation. Such events generally signify a condition such as inability to free up resources or an inconsistent view of the TIPC system.

For each TIPC event, System Viewer displays information such as the following:

- the event number and type (for example, **wvTipcStart**, **wvTipcWarning**)
- the module in which the event took place (for example, **tipcSockLib.c**)
- a tag identifying the event's location in the source code (for example, **EventTag=3**)
- the name of the function in which the event took place (for example, **Function Name=tipcSockClose**)
- relevant event parameters (for example, **Passed parameters:= P1=0x2**)

Event parameters are displayed below the function name. In some cases, to interpret a parameter you may need to consult the source code. In the source code, each System Viewer event is triggered by a call to **WV_TIPC_MARKER_x()**, where *x* is a value between 1 and 4 that indicates the number of parameters being passed.

The following are sample events logged at the VERBOSE event level. The first event, **wvTipcStart**, is an event that appears only in a VERBOSE listing:

```
3497 #: tShell10 - wvTipcStart
      wvEvtInfo:= ComponentID=windTipc Entity=WV_TIPC_SOCKET
Module=tipcSockLib.c EventTag=1 FilterID=0
      Function Name=tipcSocket
      Parameter P0=0x00000062
      Passed parameters:=
```

```
3498 #: tShell0 - wvTipcWarning
      wvEvtInfo:= ComponentID=windTipc Entity=WV_TIPC_SOCKET
Module=tipcSockLib.c EventTag=2 FilterID=0
      Function Name=tipcSocket
      Qualifier=noSock
      Passed parameters:= P1=0xffffffff9d
```

```
3511 #: tShell0 - wvTipcCritical
      wvEvtInfo:= ComponentID=windTipc Entity=WV_TIPC_SOCKET
Module=tipcSockLib.c EventTag=3 FilterID=0
      Function Name=tipcAccept
      Qualifier=noBufs
      Passed parameters:= P1=0x26bc058
```

7.4 Including TIPC System Viewer Instrumentation in a VxWorks Image Project

If your VxWorks installation includes TIPC source code, you can include TIPC System Viewer instrumentation in a build that includes the Wind River Network Stack (see the next section) and also in a build that excludes the Network Stack (see [7.4.2 Building TIPC without the Network Stack](#), p.105). When you do this, TIPC is automatically recompiled for System Viewer instrumentation; manual recompilation is not necessary.

7.4.1 Building TIPC with the Network Stack

To create a VxWorks Image Project that includes the Wind River Network Stack, TIPC system-viewer instrumentation, and System Viewer, you need to include the network stack and TIPC in your project and then include the **TIPC instrumentation** build component. The location of the **TIPC instrumentation** build component is:

**development tool components > System Viewer components >
TIPC instrumentation**

For information on including TIPC in a build, see [3. Building VxWorks to Include Wind River TIPC](#). For information on including the Network Stack in a build, see *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 1*.

7.4.2 Building TIPC without the Network Stack

You can build VxWorks to include TIPC without the Wind River Network Stack. For information, see the entry for **TIPC network stack only** in [Table 3-1](#) under [3.2 Wind River TIPC Build Components](#), p.20.

To use System Viewer with TIPC, you need to include WDB support in your image. When you build VxWorks without the network stack, you need to adhere to the following sequence of steps in Workbench, you cannot do it from the command line:

1. Exclude all networking build components from your VxWorks Image Project.

For most BSPs, a number of networking components are included in a build by default. You need to select the **Network Components** folder in the Workbench Configurator and explicitly exclude the entire folder.

2. Include the **TIPC network stack only** build component. The location of the **TIPC network stack only** build component in the Configurator is:

**Network Components > Network Protocol Components >
TIPC Components > TIPC network stack only**

3. Add WDB support to the image by including the **WDB TIPC connection** build component in your build. The location of the **WDB TIPC connection** build component in the Configurator is:

**development tool components > WDB agent components >
WDB connection > WDB TIPC connection**

4. Include the **TIPC instrumentation** build component in your build:

**development tool components > System Viewer components > TIPC in
instrumentation**

5. Build your image project.

8

Using the TIPC Test Suite

- 8.1 Introduction 107
- 8.2 Including the Test Suite in a Project 108
- 8.3 Running Tests in the Test Suite 108
- 8.4 Sample Output 111

8.1 Introduction

Wind River TIPC provides a test suite containing 15 tests that you can execute from the Workbench console or from the target shell. The tests are based on calls to the TIPC socket API. Among the tests included in the suite are tests for connection-oriented and connectionless communication, transmission of very large messages and of fragmented messages, and sending and receiving messages (for a list of the tests, see [8.3.3 Tests in the TIPC Test Suite](#), p.110). You can run the test suite, which consists of a client process and a server process, within a single node or between two nodes, which can be in the same cluster or in different clusters.

8.2 Including the Test Suite in a Project

There are three ways of gaining access to the test suite:

- Through the **TIPC test suite demo (INCLUDE_TIPC_TS)**

In this case, you can run the test suite once TIPC has started. If you want to modify or extend the source code, you can make your changes and rebuild your project. The project will be rebuilt directly from the source code. The location of the source code for the test suite is:

```
installDir/vxworks-6.x/target/src/demo/tipcTestSuite
```

- Through a VxWorks Downloadable Kernel Module Sample Project

In this case, the source code is located in your workspace. If you make changes to the code, you can recompile the test suite and then load it into the kernel.

- Through a VxWorks Real Time Process Sample Project

In this case, the source code is located in your workspace. If you make changes to the code, you can recompile the test suite and then load it into application space.

8.3 Running Tests in the Test Suite

To run tests in the test suite, you need to

1. Start the server process on a node by executing the server shell command, **tipcTS** (see [8.3.1 The tipcTS Shell Command](#), p.109).

The server must be running before you start the client process and issue a test command.

2. Start the client process and issue a test command by executing the client shell command, **tipcTC**, on a node (see [8.3.2 The tipcTC shell Command](#), p.109).

You can run the client and server processes on separate nodes or on the same node. However:

- At test time, only one server process and one client process can be running in the network.

You can test multiple nodes sequentially. For example, after running tests with the server process on one node, you can shut down the server process on that node and start it on another node.

8.3.1 The **tipcTS** Shell Command

The **tipcTS** shell command starts the test-suite server process. The syntax of the shell command is:

tipcTS *z*

where:

- *z* determines the level of verbosity:
 - 0** Non-verbose mode (the default)
 - 1** Moderate verbosity
 - 2** Highly verbose debug output

8.3.2 The **tipcTC** shell Command

The **tipcTC** shell command starts the test-suite client process, if it is not already running, and executes a test command. The syntax of the shell command is:

tipcTC *x,y,z*

where:

- *x* is an integer designating the number of the test to run (for the list of tests, see [8.3.3 Tests in the TIPC Test Suite](#), p.110).
- *y* determines whether the server process terminates after the test or keeps running:
 - 0** The server keeps running and waits for another test.
 - 1** The server shuts down.

If you do not enter a value for *z*, you do not need to enter a value for *y*. In this case, the default is **0**; the server keeps running.

- z determines the level of verbosity:
 - 0 Non-verbose mode (the default)
 - 1 Moderate verbosity
 - 2 Highly verbose debug output

For sample test output, see [8.4 Sample Output](#), p.111.

8.3.3 Tests in the TIPC Test Suite

To execute a TIPC test, you enter the **tipcTC** shell command and specify an integer representing the test you want to run. The following tests are available:

- 0 Execute test 1 through 15, in sequence.
- 1 Create a non-reliable, connectionless socket (**SOCK_DGRAM**).
- 2 Create a reliable, connectionless socket (**SOCK_RDM**).
- 3 Create a reliable, connection-oriented socket (**SOCK_STREAM**).
- 4 Create a reliable, connection-oriented socket (**SOCK_SEQPACKET**).
- 5 Shutdown a **SOCK_STREAM** connection.
- 6 Shutdown a **SOCK_SEQPACKET** connection.
- 7 Test message-size limits using **SOCK_RDM**.
- 8 Test sending of TIPC_IMPORTANCE levels with a message (see the reference page for **getsockopt()**).
- 9 Test sending TIPC socket options with a message (see the reference pages for **getsockopt()** and **setsockopt()**).
- 10 Test sending of header ancillary data with **SOCK_SEQPACKET** (reliable, connection-oriented).
- 11 Test sending of header ancillary data with **SOCK_RDM** (reliable, connectionless).
- 12 Test multicast using **SOCK_RDM**.
- 13 Test sending fragmented message using **SOCK_RDM**.
- 14 Test sending large messages (over 66000 Bytes) using **SOCK_STREAM**.
- 15 Test **sendto()** and **recvfrom()** socket routines using **SOCK_RDM**.

8.4 Sample Output

In the current release, the output from tests in the test suite is primarily useful for validating that an application and network configuration are working correctly. The failure of a test is helpful for debugging purposes, but many of the tests in the test suite, even at the most verbose level, do not provide detailed debugging information.

Output from Successful Tests (Non-Verbose)

The following shell command generates all 15 tests in the test suite, lets the server continue to run when the test complete (the default), and leaves the level of verbosity at 0 (the default).

```
-> tipcTC 0
Test # 1
TIPC connectionless (SOCK_DGRAM) test...STARTED!
TIPC connectionless (SOCK_DGRAM) test...PASSED!

Test # 2
TIPC connectionless (SOCK_RDM) test...STARTED!
TIPC connectionless (SOCK_RDM) test...PASSED!

...

Test # 14
TIPC message > 66000 bytes (SOCK_STREAM) test...STARTED!
TIPC message > 66000 bytes (SOCK_STREAM) test...PASSED!
Test # 15
TIPC sendto - recvfrom (SOCK_RDM) test...STARTED!
TIPC sendto - recvfrom (SOCK_RDM) test...PASSED!
```

Error Output from a Failed Test (Test 1; Non-Verbose)

The following shell command runs test 1 (create a non-reliable, connectionless socket), leaves the server running when the test ends, and leaves the level of verbosity at 0.

```
tipcTC 1
Test # 1
TIPC connectionless (SOCK_DGRAM) test...STARTED!
TEST FAILED sendtoSocketTIPC(): unable to allocate send buffer errno = 0:
OK
value = 0 = 0x0
```

Output from Successful Tests (Verbose, Level 1)

The following shell command generates all 15 tests in the test suite, shuts the server process down at the end of the test sequence, and sets the verbosity level to 1.

```
-> tipcTC 0,1,1
Test # 1
TIPC connectionless (SOCK_DGRAM) test...STARTED!

client_test_connectionless: subtest 1

client_test_connectionless: subtest 2
TIPC connectionless (SOCK_DGRAM) test...PASSED!

Test # 2
TIPC connectionless (SOCK_RDM) test...STARTED!

client_test_connectionless: subtest 1

client_test_connectionless: subtest 2

client_test_connectionless: subtest 3

client_test_connectionless: subtest 4
TIPC connectionless (SOCK_RDM) test...PASSED!

...

Test # 14
TIPC message > 66000 bytes (SOCK_STREAM) test...STARTED!
***** TIPC big stream test client started *****

Client: sending 75000 bytes
***** TIPC big stream test client finished *****
TIPC message > 66000 bytes (SOCK_STREAM) test...PASSED!

Test # 15
TIPC sendto - recvfrom (SOCK_RDM) test...STARTED!
TIPC sendto - recvfrom (SOCK_RDM) test...PASSED!

TIPC test suite finished
value = 0 = 0x0
```

Output from Successful Tests (Verbose, Level 2)

The following shell command generates all 15 tests in the test suite, shuts the server process down at the end of the test sequence, and sets the verbosity level to 2.

```
-> tipcTC 0,1,1
tipcTC: task spawned.  taskId=274215160
Test # 1
waiting for synchronization signal 99
```



```
acknowledging synchronization signal 99
...
got ack for synchronization signal 100
TIPC connectionless (SOCK_DGRAM) test...STARTED!

client_test_connectionless: subtest 1
client_SendConnectionless: Connectionless source: 10x100 bytes out
...
sent a message 0
sent a message 1
...
sent synchronization signal 2
got ack for synchronization signal 2
TIPC connectionless (SOCK_DGRAM) test...PASSED!
Test # 2
waiting for synchronization signal 99
acknowledging synchronization signal 99
...
TIPC connectionless (SOCK_RDM) test...STARTED!

client_test_connectionless: subtest 1
client_SendConnectionless: Connectionless source: 10x100 bytes out
waiting for synchronization signal 1
...
sent a message 98
sent a message 99
sending synchronization signal 2
sent synchronization signal 2
got ack for synchronization signal 2
TIPC connectionless (SOCK_RDM) test...PASSED!

...

est # 14
waiting for synchronization signal 99
acknowledging synchronization signal 99
...
TIPC message > 66000 bytes (SOCK_STREAM) test...STARTED!
***** TIPC big stream test client started *****

waiting for synchronization signal 1
...
acknowledged synchronization signal 1
Client: sending 75000 bytes
sent a message 0
...
got ack for synchronization signal 3
***** TIPC big stream test client finished *****
TIPC message > 66000 bytes (SOCK_STREAM) test...PASSED!
```


9

TIPC Native API

9.1	Introduction	115
9.2	Differences Between Using the Socket API and the Native API	116
9.3	Callback Routines	118
9.4	Structures for Handling Message Data	121
9.5	Routines in the TIPC Native API	121
9.6	Examples	123

9.1 Introduction

TIPC provides its own API, the TIPC native API, that you can use in place of the TIPC socket API.¹ The advantages of the TIPC native API are that it has a smaller footprint than the socket API and can improve throughput and response latencies. A disadvantage of the TIPC native API is that it is very different from the standard socket API and must be learned from scratch. In addition, the TIPC native API is only available in kernel applications. RTPs must use the TIPC socket API.

-
1. The TIPC native API has not been finalized by the TIPC Working Group of the Multicore Association (see <http://www.multicore-association.org>) and is still subject to change.

The socket API and the native API are not mutually exclusive. If the TIPC socket API is included in a build, you can use both the socket API and the native API in a kernel application.

9.2 Differences Between Using the Socket API and the Native API

There are a number of important conceptual differences between programming with the native API and programming with the socket API.

Ports

The fundamental communication endpoint of the native API is a *port*, which operates at a more primitive level than a socket. Applications that use the native API with TIPC ports are sometimes required to deal with aspects of the TIPC protocol that are hidden by the socket API, including handling undeliverable messages that are returned to the sending port and managing the handshaking required to set up and tear down port-to-port connections. Unlike the socket API, the native API recognizes only two fundamental types of ports, connection-oriented and connectionless ports.

Accessing a specific port via the native API is not re-entrant. This is because the header for each message is cached in the port structure and is not locked. Parallel access can result in corrupted header information. Applications need to take account of this.

Port Reference

Every TIPC port has a unique *reference* value that is analogous to the file descriptor value that is associated with a socket. Native API routines that manipulate ports use the port reference argument to identify the port, rather than a pointer to the actual port data structure; this allows TIPC to gracefully handle cases where an application inadvertently attempts to utilize a port that no longer exists.

User Registration

TIPC allows an application using the native API to register as a user, and assigns it a user identifier (see **`tipc_attach()`** in [C. TIPC Native Routines](#)). If this user identifier is provided by the application when it creates a port, TIPC deletes the port automatically if the application later deregisters itself. This feature can

simplify things for a programmer whose application uses a constantly changing set of ports, since TIPC takes care of deleting all ports currently in use by the application when the application terminates. (Applications not wishing to take advantage of this capability can skip the optional registration process entirely and simply create their ports anonymously using a user identifier value of 0.)

Sending Messages

Applications can send messages using the native API in much the same way as with the socket API. The message can be specified either as a set of one or more byte arrays (using the **iovec** structure) or as a socket buffer (using the **sk_buff** structure), as long as it does not exceed TIPC's 66000 byte limit on message size (for information on the **iovec** and **sk_buff** structures, see [9.4 Structures for Handling Message Data](#), p.121). The latter form can improve performance by eliminating the need for TIPC to copy the data into a socket buffer, but only if the application that creates the buffer reserves 80 bytes of headroom to allow a TIPC message header and data link header to be prepended easily.

Receiving messages

The native API does not provide a synchronous mechanism for receiving messages sent to a port; there is no equivalent of the **recv()**, **recvfrom()**, or **recvmsg()** routines that the socket API provides. Instead, an application specifies a set of message handling callback routines when it creates a port; TIPC then invokes the appropriate routine each time a message is received by the port.

SMP

On SMP systems, when running multiple links between nodes, there is a race condition when the discovery mechanism detects the new node and tries to create a link. A check is done to determine if the node is already known, and then the node structure is allocated and added to the list of known nodes. If the second link detects the same new node and checks for this node after the first link has checked for the node but before the first link has created the node structure, then the duplication of node structures can result in an error.

9.3 Callback Routines

The native API provides callback routines that a user can implement for detecting a change in TIPC's operating state and for handling messages at a specific port. The following callbacks are available:

Table 9-1 **Callbacks in the TIPC Native API**

Callback	Comment
tipc_mode_event	<p>For handling changes in TIPC's operating state. In the current release, the possible operating modes are:</p> <ul style="list-style-type: none">▪ TIPC_NOT_RUNNING▪ TIPC_NODE_MODE The node does not have a TIPC address, and is in standalone mode.▪ TIPC_NET_MODE The node has a TIPC address and is part of a TIPC network. <p>For more information, see tipc_mode_event under tipc_attach() in <i>C. TIPC Native Routines</i>.</p>
tipc_msg_event()	<p>For handling incoming messages on a specific port. For more information, see tipc_msg_event under tipc_createport() in <i>C. TIPC Native Routines</i>.</p>
tipc_named_msg_event()	<p>For handling incoming messages sent to a port name or port-name sequence.</p> <p>For more information, see tipc_named_msg_event under tipc_createport() in <i>C. TIPC Native Routines</i>. For a sample implementation, see named_msg_event() under <i>9.6 Examples</i>, p.123.</p>

Table 9-1 Callbacks in the TIPC Native API

Callback	Comment
tipc_conn_msg_event()	For handling incoming connection-oriented messages. For more information, see tipc_conn_msg_event under tipc_createport() in <i>C. TIPC Native Routines</i> . For a sample implementation, see mon_conn_msg_event_cb() under <i>9.6 Examples</i> , p. 123.
tipc_continue_event()	For handling congestion abatement on a specific port. This callback is only called if a previous send failed due to congestion.. For more information, see tipc_continue_event under tipc_createport() in <i>C. TIPC Native Routines</i> .
tipc_msg_err_event()	For handling for handling messages with an error code, such as rejected messages, that have been sent to a specific port ID (see <i>2.4 Message Reliability and Rejected Messages</i> , p. 11). For more information, see tipc_msg_err_event under tipc_createport() in <i>C. TIPC Native Routines</i> .

Table 9-1 **Callbacks in the TIPC Native API**

Callback	Comment
tipc_named_msg_err_event()	For handling for handling messages with an error code, such as rejected messages, that have been sent to a port name or port-name sequence (see 2.4 Message Reliability and Rejected Messages , p.11). For more information, see tipc_named_msg_err_event under tipc_createport() in C. TIPC Native Routines .
tipc_conn_shutdown_event	For handling rejected connection messages (see 2.4 Message Reliability and Rejected Messages , p.11). For more information, see tipc_conn_shutdown_event under tipc_createport() in C. TIPC Native Routines . For a sample implementation, see mon_shutdown_cb() under 9.6 Examples , p.123.

If TIPC receives a message for which no callback routine has been specified, it automatically rejects the message or, if the message was an error message, discards it.

TIPC callback routines execute in a TIPC kernel thread, rather than one of the application's threads, and must be non-blocking. Callbacks must either handle critical section issues that arise between threads or transfer responsibility to an application thread, as outlined in the example given for handling a synchronous receive (see [9.6.3 Receiving a Synchronous Message](#), p.125), thereby allowing the application to emulate a synchronous receive capability of its own.

9.4 Structures for Handling Message Data

The TIPC native API uses the following structure, comparable to a socket buffer, for sending and receiving data:

```
struct sk_buff {
    struct sk_buff *next;           / * ptr to next buffer in list * /
    struct sk_buff *prev;          / * ptr to previous buffer in list * /
    M_BLK_ID      mBlkId;         / * ptr to mBlk * /
    char cb[sizeof(struct tipc_skb_cb)]; / * control block area * /
    unsigned int   len;
    unsigned char  *data;
    unsigned char  *tail;
};
```

The **sk_buff** structure contains a **tipc_skb_cb** structure and an **iovec** structure.

The definition of the **tipc_skb_cb** structure, which is for control blocks, is:

```
struct tipc_skb_cb {
    void *handle;
};
```

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

```
/*
 * VxWorks iovec structure (treat as an array for multiple segments)
 */
struct iovec {
    char  *iov_base;      / * Base address. * /
    size_t iov_len;       / * Length. * /
};
```

9.5 Routines in the TIPC Native API

This section lists and gives brief descriptions of the routines in the TIPC native API. For more information on the routines, see [C. TIPC Native Routines](#).

TIPC operating mode routines:

tipc_get_addr()	Get Z.C.N of own node
tipc_get_mode()	Get TIPC operating mode
tipc_attach()	Register application as a TIPC user
tipc_detach()	Deregister TIPC user & free all associated ports

TIPC port manipulation routines:

tipc_createport()	Create a TIPC port and generate reference
tipc_deleteport()	Delete a TIPC port and obsolete reference
tipc_ref_valid()	Determine whether port reference is valid
tipc_ownidentity()	Get port ID of port
tipc_set_portimportance()	Set port traffic importance level
tipc_portimportance()	Get port traffic importance level
tipc_set_portunreliable()	Set port traffic source droppable setting
tipc_portunreliable()	Get port traffic source droppable setting
tipc_set_portunreturnable()	Set port traffic destination droppable setting
tipc_portunreturnable()	Get port traffic destination droppable setting
tipc_publish	Bind name/name sequence to port
tipc_withdraw()	Unbind name/name sequence from port
tipc_connect2port()	Associate port with peer
tipc_disconnect()	Disassociate port from peer
tipc_shutdown()	Shut down connection to peer and disassociate
tipc_isconnected()	Determine whether port is currently connected
tipc_peer()	Get port ID of peer port

TIPC messaging routines:

tipc_send()	Send iovec(s) on connection
tipc_send_buf()	Send sk_buff on connection
tipc_send2name()	Send iovec(s) to port name
tipc_send_buf2name()	Send sk_buff to port name
tipc_send2port()	Send iovec(s) to port ID
tipc_send_buf2port()	Send sk_buff to port ID
tipc_multicast()	Multicast iovec(s) to port

TIPC subscription routine:

tipc_ispublished() Determines whether a specific name has been published

9.6 Examples

This section provides examples of the following:

- [9.6.1 Performing Basic Port Operations](#), p.123
- [9.6.2 Registering a TIPC User](#), p.124
- [9.6.3 Receiving a Synchronous Message](#), p.125
- [9.6.4 Using the TIPC Topology Service](#), p.126

You can find demonstration programs that use the native API at <http://tipc.sf.net>. In addition, the TIPC source code contains sections that use the native API in the same way that an application can. Examples are:

- **tipc_cfg_init()** in **net/tipc/config.c**
net/tipc/config.c contains the TIPC configuration service (using port name {0,Z.C.N}), which handles messages sent by the **tipcConfig** application. It uses a connectionless request-and-reply approach to messaging.
- **tipc_subscr_start()** in **net/tipc/subscr.c**
net/tipc/subscr.c file contains the TIPC topology service (using port name {1,1}), which handles subscription requests from applications and returns subscription events. It demonstrates the way to handle connection establishment (both explicit and implied) and tear-down (both self-initiated and peer-initiated).

9.6.1 Performing Basic Port Operations

Create a port:

```
static u32 port_ref;  
  
tipc_createport(0, NULL, TIPC_LOW_IMPORTANCE,  
               NULL, NULL, NULL,  
               NULL, named_msg_event, NULL,  
               NULL, &port_ref);
```

Bind the port name {100,123} with cluster scope to the port:

```
struct tipc_name_seq seq;

seq.type = 100 ;
seq.lower = 123 ;
seq.upper = 123 ;
tipc_publish(port_ref, TIPC_CLUSTER_SCOPE, &seq);
```

Process messages sent to the port {100,123}:

```
/* Note: This callback routine was specified during port creation above */
/

static void named_msg_event(void *usr_handle,
    u32 port_ref,
    struct sk_buff **buf,
    unsigned char const *data,
    unsigned int size,
    unsigned int importance,
    struct tipc_portid const *orig,
    struct tipc_name_seq const *dest)
{
    /* * data points to message content, size indicates how much */

    printf("%s", data);

    /* * can send reply message(s) back to originator, if desired */

    struct iovec my_iov;
    char reply_info[30];

    strcpy(reply_info, "here is the reply");
    my_iov.iov_base = reply_info;
    my_iov.iov_len = strlen(reply_info) + 1;
    tipc_send2port(port_ref, orig, 1, &my_iov);

    /* TIPC discards the received message upon exit */
}
```

Delete the port:

```
tipc_deleteport(port_ref);
```

9.6.2 Registering a TIPC User

Register a TIPC user:

```
static u32 user_ref;

tipc_attach(&user_ref, NULL, NULL);
```

Create a port and associate it with a registered TIPC user:

```
static u32 port_ref;

tipc_createport(user_ref, NULL, TIPC_LOW_IMPORTANCE,
               NULL, NULL, NULL,
               NULL, named_msg_event, NULL,
               NULL, &port_ref);
```

Deregister a TIPC user (and all associated ports):

```
tipc_detach(user_ref);
```

9.6.3 Receiving a Synchronous Message

Application thread:

```
/* use one definition for either events or semaphores */
#define GET_TRIGGER eventReceive(VXEV01, EVENTS_WAIT_ALL, WAIT_FOREVER, N
ULL)
#define GET_TRIGGER semTake(semTaskSend)

/* Initialize data structures for holding ingress queue */

struct sk_buff_head message_q;
wait_queue_head_t wait_q;

skb_queue_head_init(&message_q);
init_waitqueue_head(&wait_q);

/* Wait for messages; process & discard each one in turn */

while (1) {
    struct sk_buff *skb;

    GET_TRIGGER;

    skb = skb_dequeue(&message_q);

    < ... Process message as required ... >

    kfree_skb(skb);
}
```

Callback routine that converts an asynchronous receive into a synchronous receive:

```
/* use one definition for either events or semaphores */
#define SEND_TRIGGER eventSend(callTask, VXEV01)
#define SEND_TRIGGER semGive(semTaskSend)
static void named_msg_event
(
    void *usr_handle,
    u32 port_ref,
    struct sk_buff **buf,
    unsigned char const *data,
    unsigned int size,
    unsigned int importance,
    struct tipc_portid const *orig,
    struct tipc_name_seq const *dest
)
{
    /* Add message to queue of unprocessed messages */

    skb_queue_tail(&message_q, *buf);

    /* Tell TIPC *not* to discard the received message upon exit */

    *buf = NULL;

    /* Wake up application */

    SEND_TRIGGER;
}
```

9.6.4 Using the TIPC Topology Service

The following example creates a connection to the TIPC topology server to monitor a TIPC name. When the name appears, the callback sends a signal to the calling task. When the name disappears, the callback registered closes down the connection to the topology server and sends a signal to the calling task.

```
#define GET_TRIGGER eventReceive(VXEV01 | VXEV02, EVENTS_WAIT_ANY, WAIT_
FOREVER, NULL)
#define SEND_TRIGGER eventSend(callTask, VXEV01);
#define SEND_TERMINATE eventSend(callTask, VXEV02);
#define TIPC_EXPERIMENT_TYPE 1000 / * Any number * /
#define TIPC_EXPERIMENT_INSTANCE 100 / * Any number * /

int callTask; /* task that needs to be signalled */
```

```

/*****
 *
 * mon_shutdown_cb - handle connection termination message
 *
 * Used in this code to receive any topology server message based
 * on monitoring of the relevant {type,instance}.
 *
 * RETURNS:  N/A
 */
static void mon_shutdown_cb(void *usr_handle,
    u32 port_ref,
    struct sk_buff **buf,
    unsigned char const *data,
    unsigned int size,
    int reason)
{
    /* TIPC has already disconnected port, so just delete it */
    tipc_deleteport(port_ref);

    /* wake up any application routine to let it gracefully exit */
    SEND_TERMINATE;
}
/*****
 *
 * mon_conn_msg_event_cb-call back for connection oriented message event
 *
 * Used in this code to receive any topology server message based
 * on monitoring of the relevant {type,instance}.
 *
 * where:
 *     buf = data packet
 *     size = number of bytes in message
 *
 * RETURNS:  N/A
 */
static void mon_conn_msg_event_cb(void *usr_handle,
    u32 port_ref,
    struct sk_buff **buf,
    unsigned char const *data,
    unsigned int size,
    unsigned int importance,
    struct tipc_portid const *orig,
    struct tipc_name_seq const *destination)
{
    struct tipc_event * event;      /* topology events (subscription) */
    int res;                       /* result of an operation */

    event = (struct tipc_event *)data; /* point event to data */

    if (event->event==TIPC_SUBSCR_TIMEOUT) /* timed out subscription */
    {
        goto monitorExit;
    }
    if (event->event == TIPC_WITHDRAWN) /* withdrawn subscription */
    {

```

```
        goto monitorExit;
    }
    if (event->event == TIPC_PUBLISHED)          /* publication detected */
    {
        SEND_TRIGGER;
    }

    /* keep monitoring for a withdrawl */
    return;

monitorExit:
    res = tipc_shutdown(port_ref);
    if (res)
    {
        /* handle failure to shutdown */
    }

    /* wake up application to let it gracefully exit */
    SEND_TERMINATE;

    res = tipc_deleteport(port_ref);
    if (res)
    {
        /* handle failure to delete port */
    }

    return;
}
/*****
 *
 * monitorPublication - Monitor a publication of the receiver of msgs
 *
 * The monitor sets up a connection to the topology server using Native
 * API. When publication is detected, the callback signals the calling
 * Task. When the publication is revoked or the subscription times out,
 * the monitoring stops by closing the connection to the topology server.
 *
 * Note you may have to ensure that the calling task is pending on an
 * eventReceive before the subscription is created to ensure an event is
 * not missed (not addressed in this code).
 *
 * monitorPublication (int callingTask)
 *
 * where:
 *     callingTask = Task ID of caller
 *
 * RETURNS:  STATUS
 */
```



```

STATUS monitorPublication (int callingTask) {
    struct tipc_subscr subscr;          /* subscription of interest */
    struct tipc_name name;              /* name to send to */
    int res;                            /* result of operations */
    u32 port_ref;                       /* port for communications to top srv */
    u32 user_ref;                       /* user reference */
    struct iovec msg_sect;              /* iovec for data */

    callTask = callingTask;
    /* set up subscription */
    subscr.seq.type = TIPC_EXPERIMENT_TYPE;
    subscr.seq.lower = TIPC_EXPERIMENT_INSTANCE;
    subscr.seq.upper = TIPC_EXPERIMENT_INSTANCE;
    subscr.timeout = TIPC_WAIT_FOREVER;
    subscr.filter = TIPC_SUB_PORTS;

    /* set up addressing */
    name.type = TIPC_TOP_SRV;
    name.instance = TIPC_TOP_SRV;

    res = tipc_attach(&user_ref, NULL, NULL);
    if (res)
    {
        printf("monitorPublication: tipc_attach returned %d errno=%d)\n",
            res, errno);
        return ERROR;
    }

    res = tipc_createport(user_ref, NULL, TIPC_LOW_IMPORTANCE,
        NULL, NULL, mon_shutdown_cb,
        NULL, NULL,
        mon_conn_msg_event_cb, NULL, &port_ref);

    if (res)
        return res;

    /* send first message to port */
    msg_sect.iov_base = (char *)&subscr;
    msg_sect.iov_len = sizeof(subscr);

    tipc_send2name(port_ref,
        &name,
        0 /* domain of 0:own zone */,
        1 /* num_sect */,
        msg_sect);
    return OK;
}

```


A

Libraries

tipc_config_show	- Library of Wind River TIPC Config and Management cmds	131
tipc_lib	- Library of Wind River TIPC socket-based and utility routines in Kernel space	132
tipc_native	- Library of Wind River TIPC Native API in Kernel space.....	133

tipc_config_show

NAME	tipc_config_show – Library of Wind River TIPC Config and Management cmds
ROUTINES	tipcConfig() – the public API for TIPC configuration and management commands tipcSysPoolShow() – display TIPC system-pool statistics tipcDataPoolShow() – display TIPC data-pool statistics
DESCRIPTION	<p>This library contains the TIPC Configuration and Management commands. It is largely based on the tipc-config utility in Linux.</p> <p>In the current design, these commands are only accessible from a kernel context. RTP support is not included.</p> <p>Many of the public management commands are available to be used with remote nodes once a remote node is identified.</p>
INCLUDE FILES	none

tipc_lib

NAME `tipc_lib` – Library of Wind River TIPC socket-based and utility routines in Kernel space

ROUTINES

- `accept()` – accept a request for a connection to a socket
- `bind()` – bind an address to a socket
- `close()` – close a socket
- `connect()` – request a connection to a socket
- `getpeername()` – get the port ID of a peer socket
- `getsockname()` – get the port ID of a socket
- `getsockopt()` – get the value of an option associated with a socket
- `listen()` – enable a socket to receive connection requests
- `recv()` – receive data from a socket
- `recvfrom()` – receive data from a socket
- `recvmsg()` – receive data from a socket
- `send()` – send a message to a socket
- `sendmsg()` – send a message to a socket
- `sendto()` – send a message to a socket
- `setsockopt()` – set the value of an option associated with a socket
- `shutdown()` – shut down a connection
- `socket()` – create a socket
- `tipc_addr()` – combine zone, cluster, and node numbers into a TIPC address
- `tipc_cluster()` – take a TIPC network address and return the cluster number
- `tipc_node()` – take a TIPC address and return the node number
- `tipc_zone()` – take a TIPC address and return the zone number

DESCRIPTION This library contains TIPC socket-based routines and utility routines for handling TIPC addresses.

Many of the socket routines make reference to a **sockaddr** structure that usually refers to the **sockaddr_tipc** structure used by the TIPC code. The **sockaddr_tipc** structure is shown below.

```
struct sockaddr_tipc {
    unsigned char  addrlen;           /* 16 */
    unsigned char  family;            /* AF_TIPC */
    unsigned char  addrtype;          /* TIPC_ADDR_XXX */
    unsigned char  scope;             /* used with bind */
    union {
        struct tipc_portid id;        /* if TIPC_ADDR_ID */
        struct tipc_name_seq nameseq; /* if TIPC_ADDR_NAMESEQ/_MCAST */
        struct {
            struct tipc_name name;
            __u32 domain;           /* 0: own zone; used w/ connect,
sendto */
        } name;
    } addr;
};
```

INCLUDE FILES **tipc/tipc.h**

SEE ALSO *Wind River TIPC for VxWorks 6 Programmer's Guide*

tipc_native

NAME **tipc_native** – Library of Wind River TIPC Native API in Kernel space

ROUTINES

tipc_attach() – Register a TIPC user (native API only)
tipc_connect2port() – Associate a TIPC port with its peer (native API only)
tipc_createport() – Create a TIPC port (native API only)
tipc_deleteport() – Delete a TIPC port (native API only)
tipc_detach() – Unregister a TIPC user (native API only)
tipc_disconnect() – Disassociate a TIPC port with its peer (native API only)
tipc_forward2name() – Forward a message to the named port (native API only - may be
 obsoleted)
tipc_forward2port() – Forward a message to a port (native API only - may be obsoleted)
tipc_forward_buf2name() – Forward a buffer to the named port (native API only - may be
 obsoleted)
tipc_forward_buf2port() – Forward a buffer to a port (native API only - may be obsoleted)
tipc_get_addr() – Get the network address for this node (native API only)
tipc_get_mode() – Get operating mode of TIPC (native API only)
tipc_isconnected() – Determine if a TIPC port is connected (native API only)
tipc_ispublished() – Determine if a TIPC name exists (native API only)
tipc_multicast() – Multicast data to a set of named TIPC ports (native API only)
tipc_ownidentity() – Get port ID of TIPC port (native API only)
tipc_peer() – Get the port ID of a TIPC port's peer (native API only)
tipc_portimportance() – Get importance of TIPC port messages (native API only)
tipc_portunreliable() – Get reliability of TIPC port messages (native API only)
tipc_portunreturnable() – Get returnability of TIPC port messages (native API only)
tipc_publish() – Add a name or name sequence to a TIPC port (native API only)
tipc_ref_valid() – Validate a reference to a TIPC port (native API only)
tipc_send() – Send data over TIPC connection (native API only)
tipc_send2name() – Send data to a named TIPC port (native API only)
tipc_send2port() – Send data to a TIPC port ID (native API only)
tipc_send_buf() – Send message buffer over TIPC connection (native API only)
tipc_send_buf2name() – Send message buffer to a named TIPC port (native API only)
tipc_send_buf2port() – Send message buffer to a TIPC port ID (native API only)
tipc_set_portimportance() – Set importance of TIPC port messages (native API only)
tipc_set_portunreliable() – Set reliability of TIPC port messages (native API only)

tipc_set_portunreturnable() – Set returnability of TIPC port messages (native API only)
tipc_shutdown() – Disconnect a TIPC port from its peer (native API only)
tipc_withdraw() – Remove a name or name sequence from a TIPC port (native API only)

DESCRIPTION

This library contains TIPC native API routines for handling TIPC functionality.

The TIPC native API is only available to kernel applications and cannot be used from an RTP. The native API is a different method of communicating with TIPC that is distinct from the Socket API, the more commonly used TIPC API.

The native API relies on providing TIPC with a number of callback routines that are called for events that occur. The native API application needs to supply these routines and register them as a port is created.

The TIPC native API allows programmers to access the capabilities of TIPC in a more direct manner than with the socket API. As such, more care may be required than when using the socket API. See some caveats and warnings at the bottom of this section.

Benefits of the native API:

1. Low-level operation can lead to faster execution speed.
2. Can exclude socket code from system to reduce object code size.

Limitations of the native API:

1. Not available to user-space applications.
2. Low-level operation places a greater burden on programmer.

Concepts

There are a number of important conceptual differences between programming with the native API and programming with the socket API. Understanding these concepts is an essential pre-requisite for using the native API effectively.

Ports:

The fundamental communication endpoint of the native API is a "port", which operates at a much more primitive level than a socket. Applications using TIPC ports are sometimes required to deal with aspects of the TIPC protocol that were hidden by the socket API, including handling undeliverable messages that are returned to the sending port and managing the handshaking required to set up and tear down port-to-port connections. And unlike the socket API, there are only two fundamental types of ports which are connection-oriented and connectionless ports.

Port reference:

Every TIPC port has a unique "reference" value, which is analogous to the file descriptor value that is associated with a socket. Native API routines that manipulate ports use the

port reference argument to identify the port, rather than a pointer to the actual port data structure; this allows TIPC to gracefully handle cases where an application inadvertently attempts to utilize a port that no longer exists.

User registration:

TIPC allows an application using the native API to register as a "user", and assigns it a user identifier. If this user identifier is provided by the application when it creates a port, TIPC will delete the port automatically if the application later deregisters itself. This feature can simplify things for a programmer whose application uses a constantly changing set of ports, since TIPC takes care of deleting all ports currently in use by the application when the application terminates. (Applications not wishing to take advantage of this capability can skip the optional registration process entirely and simply create their ports anonymously using a user identifier value of 0.)

Sending messages:

Applications can send messages using the native API in much the same way as with the socket API. The message can be specified either as a set of one or more byte arrays (using the "iovec" structure) or as a socket buffer (using the "sk_buff" structure), as long as it does not exceed TIPC's 66000 byte limit on message size. The latter form can improve performance by eliminating the need for TIPC to copy the data into a socket buffer but only if the application that creates the buffer reserves 80 bytes of headroom to allow a TIPC message header and data link header to be prepended easily.

Receiving messages:

The native API does not provide any synchronous mechanism for receiving messages sent to a port. (That is, there is no equivalent of the **recv()**, **recvfrom()**, or **recvmsg()** routines that the socket API provides.) Instead, an application specifies a set of message handling callback routines when it creates a port; TIPC then invokes the appropriate routine each time a message is received by the port.

Individual callback routines may be specified to handle:

- 1) a direct message (i.e. one sent to a port ID)
- 2) a named message (i.e. one sent to a port name or name sequence)
- 3) a connection message (i.e. one sent on an established connection)
- 4) an errored direct message (i.e. a direct message that was returned)
- 5) an errored named message (i.e. a named message that was returned)
- 6) an errored connection message (i.e. a connection message that was returned)

An application only needs to supply callback routines for the messages that the port actually needs to handle. If TIPC receives a message for which no callback routine has been specified, it automatically rejects the message (or, in the case of an errored message, discards it).

Since the callback routine executes in a TIPC kernel thread, rather than one of the application's threads, the programmer must be prepared to handle any critical section

issues that arise between the various threads. Alternatively, the callback routine can transfer responsibility to an application thread (as outlined in the example for handling a synchronous receive), thereby allowing the application to emulate a synchronous receive capability of its own. Finally, since the callback routines execute in a TIPC kernel thread, they must be non-blocking.

Routines

The native API routines listed below are available to programmers. More detail about the arguments and return value for each of these routines can be found by looking at the function prototypes in **tipc.h** as well as the later portion of this section. In many cases the use of the routine will be obvious. You can also consult the examples section below and/or the source code for each routine to learn more about what these routines do and how to use them.

WARNING! The native API is still under development at this time
WARNING! and has not been finalized by the TIPC Working Group. Expect
WARNING! changes in future versions of TIPC.

/* TIPC operating mode routines */

tipc_get_addr() - get Z.C.N of own node
tipc_get_mode() - get TIPC operating mode
tipc_attach() - register application as a TIPC user
tipc_detach() - deregister TIPC user & free all associated ports

/* TIPC port manipulation routines */

tipc_createport() - create a TIPC port & generate reference
tipc_deleteport() - delete a TIPC port & obsolete reference
tipc_ref_valid() - determine if port reference is valid
tipc_ownidentity() - get port ID of port
tipc_set_portimportance() - set port traffic importance level
tipc_portimportance() - get port traffic importance level
tipc_set_portunreliable() - set port traffic "source droppable" setting
tipc_portunreliable() - get port traffic "source droppable" setting
tipc_set_portunreturnable() - set port traffic "destination droppable" setting
tipc_portunreturnable() - get port traffic "destination droppable" setting
tipc_publish() - bind name/name sequence to port
tipc_withdraw() - unbind name/name sequence from port
tipc_connect2port() - associate port with peer
tipc_disconnect() - disassociate port with peer
tipc_shutdown() - shut down connection to peer & disassociate

tipc_isconnected() - determine if port is currently connected
tipc_peer() - get port ID of peer port

/ TIPC messaging routines */*

tipc_send() - send iovec(s) on connection
tipc_send_buf() - send sk_buff on connection
tipc_send2name() - send iovec(s) to port name
tipc_send_buf2name() - send sk_buff to port name
tipc_send2port() - send iovec(s) to port ID
tipc_send_buf2port() - send sk_buff to port ID
tipc_multicast() - multicast iovec(s) to port name sequence

tipc_forward2name() - [may be obsoleted]
tipc_forward_buf2name() - [may be obsoleted]
tipc_forward2port() - [may be obsoleted]
tipc_forward_buf2port() - [may be obsoleted]

/ TIPC subscription routines */*

tipc_ispublished() - determines if a specific name has been published
tipc_available_nodes() - [likely to be obsoleted - not described further]

Examples

Basic port operations

Create a port:

```
static u32 port_ref;
```

```
tipc_createport(0, NULL, TIPC_LOW_IMPORTANCE,
               NULL, NULL, NULL,
               NULL, named_msg_event, NULL,
               NULL, &port_ref);
```

Bind the name {100,123} with "cluster" scope to the port:

```
struct tipc_name_seq seq;

seq.type = 100 ;
seq.lower = 123 ;
seq.upper = 123 ;
tipc_publish(port_ref, TIPC_CLUSTER_SCOPE, &seq);
```

Process messages sent to port {100,123}:

```
/* Note: This callback routine was specified during port creation above */

static void named_msg_event(void *usr_handle,
                           u32 port_ref,
                           struct sk_buff **buf,
                           unsigned char const *data,
                           unsigned int size,
                           unsigned int importance,
                           struct tipc_portid const *orig,
                           struct tipc_name_seq const *dest)
{
    /* data points to message content, size indicates how much */

    printf("%s", data);

    /* can send reply message(s) back to originator, if desired */

    struct iovec my_iov;
    char reply_info[30];

    strcpy(reply_info, "here is the reply");
    my_iov.iov_base = reply_info;
    my_iov.iov_len = strlen(reply_info) + 1;
    tipc_send2port(port_ref, orig, 1, &my_iov);

    /* TIPC discards the received message upon exit */
}
```

Delete the port:

```
tipc_deleteport(port_ref);
```

TIPC user registration

Register TIPC user:

```
static u32 user_ref;
```

```
tipc_attach(&user_ref, NULL, NULL);
```

Create port and associate with registered TIPC user:

```
static u32 port_ref;
```

```
tipc_createport(user_ref, NULL, TIPC_LOW_IMPORTANCE,
                NULL, NULL, NULL,
                NULL, named_msg_event, NULL,
                NULL, &port_ref);
```

Deregister TIPC user (and all associated ports):

```
tipc_detach(user_ref);
```

Synchronous message receive

Application thread:

```
/* use one definition for either events or semaphores */
#define GET_TRIGGER eventReceive(VXEV01, EVENTS_WAIT_ALL, WAIT_FOREVER, N
ULL)
#define GET_TRIGGER semTake(semTaskSend)

/* Initialize data structures for holding ingress queue */

struct sk_buff_head message_q;
wait_queue_head_t wait_q;
```

```
skb_queue_head_init(&message_q);
init_waitqueue_head(&wait_q);

/ * Wait for messages; process & discard each one in turn * /

while (1) {
    struct sk_buff *skb;

    GET_TRIGGER;

    skb = skb_dequeue(&message_q);

    < ... Process message as required ... >

    kfree_skb(skb);
}
```

Callback routine converts asynchronous receive into synchronous receive:

```
/ * use one definition for either events or semaphores * /
#define SEND_TRIGGER eventSend(callTask, VXEVO1)
#define SEND_TRIGGER semGive(semTaskSend)

static void named_msg_event(void *usr_handle,
    u32 port_ref,
    struct sk_buff **buf,
    unsigned char const *data,
    unsigned int size,
    unsigned int importance,
    struct tipc_portid const *orig,
    struct tipc_name_seq const *dest)
{
    / * Add message to queue of unprocessed messages * /

    skb_queue_tail(&message_q, *buf);

    / * Tell TIPC *not* to discard the received message upon exit * /

    *buf = NULL;
```

```

    /* Wake up application */

    SEND_TRIGGER;
}

```

Topology Service Usage

This example creates a connection to the topology server to monitor a TIPC name. When the name appears, the callback will send a signal to the calling task. When the name disappears, the callback registered will close down the connection to the topology server and send a signal to the calling task.

```

#define GET_TRIGGER eventReceive(VXEV01 | VXEV02, EVENTS_WAIT_ANY, WAIT_FOR,
REVER, NULL)
#define SEND_TRIGGER eventSend(callTask, VXEV01);
#define SEND_TERMINATE eventSend(callTask, VXEV02);
#define TIPC_EXPERIMENT_TYPE 1000 /* Any number */
#define TIPC_EXPERIMENT_INSTANCE 100 /* Any number */

int callTask; /* task that needs to be signalled */

/ *****
*
* mon_shutdown_cb - handle connection termination message
*
* Used in this code to receive any topology server message based
* on monitoring of the relevant {type,instance}.
*
* RETURNS: N/A
* /

static void mon_shutdown_cb(void *usr_handle,
                           u32 port_ref,
                           struct sk_buff **buf,
                           unsigned char const *data,
                           unsigned int size,
                           int reason)
{
    /* TIPC has already disconnected port, so just delete it */
    tipc_deleteport(port_ref);
}

```

```
    / * wake up any application routine to let it gracefully exit * /  
    SEND_TERMINATE;  
}  
  
/ *****  
*  
* mon_conn_msg_event_cb - call back for a connection oriented message event  
*  
* Used in this code to receive any topology server message based  
* on monitoring of the relevant {type,instance}.  
*  
* where:  
*   buf = data packet  
*   size = number of bytes in message  
*  
* RETURNS: N/A  
* /  
  
static void mon_conn_msg_event_cb(void *usr_handle,  
                                u32 port_ref,  
                                struct sk_buff **buf,  
                                unsigned char const *data,  
                                unsigned int size,  
                                unsigned int importance,  
                                struct tipc_portid const *orig,  
                                struct tipc_name_seq const *destination)  
{  
    struct tipc_event * event;    / * topology events (subscription) * /  
    int res;    / * result of an operation * /  
  
    event = (struct tipc_event *)data; / * point event to data * /  
  
    if (event->event == TIPC_SUBSCR_TIMEOUT) / * timed out subscription * /  
    {  
        goto monitorExit;  
    }  
    if (event->event == TIPC_WITHDRAWN) / * withdrawn subscription * /  
    {  
        goto monitorExit;  
    }  
    if (event->event == TIPC_PUBLISHED) / * publication detected * /  
    {  
        SEND_TRIGGER;  
    }  
}
```

```

    }

    /* keep monitoring for a withdrawl */
    return;

monitorExit:
    res = tipc_shutdown(port_ref);
    if (res)
    {
        /* handle failure to shutdown */
    }

    /* wake up application to let it gracefully exit */
    SEND_TERMINATE;

    res = tipc_deleteport(port_ref);
    if (res)
    {
        /* handle failure to delete port */
    }

    return;
}

/ *****
*
* monitorPublication - Monitor a publication of the receiver of msgs
*
* The monitor sets up a connection to the topology server using the Native
* API. When the publication is detected, the callback signals the calling
* Task. When the publication is revoked or the subscription times out,
* the monitoring stops by closing the connection to the topology server.
*
* Note you may have to ensure that the calling task is pending on an
* eventReceive before the subscription is created to ensure an event is
* not missed (not addressed in this code).
*
* monitorPublication (int callingTask)
*
* where:
*   callingTask = Task ID of caller
*
* RETURNS: STATUS

```

* /

```
STATUS monitorPublication (int callingTask) {
    struct tipc_subscr subscr;    /* subscription of interest */
    struct tipc_name  name;      /* name to send to */
    int              res;        /* result of operations */
    u32              port_ref;    /* port for communications to top srv */
    u32              user_ref;    /* user reference */
    struct iovec      msg_sect;   /* iovec for data */

    callTask = callingTask;
    /* set up subscription */
    subscr.seq.type = TIPC_EXPERIMENT_TYPE;
    subscr.seq.lower = TIPC_EXPERIMENT_INSTANCE;
    subscr.seq.upper = TIPC_EXPERIMENT_INSTANCE;
    subscr.timeout = TIPC_WAIT_FOREVER;
    subscr.filter = TIPC_SUB_PORTS;

    /* set up addressing */
    name.type = TIPC_TOP_SRV;
    name.instance = TIPC_TOP_SRV;

    res = tipc_attach(&user_ref, NULL, NULL);
    if (res)
    {
        printf("monitorPublication: tipc_attach returned %d (errno=%d)\n", res, errno);
        return ERROR;
    }

    res = tipc_createport(user_ref, NULL, TIPC_LOW_IMPORTANCE,
        NULL, NULL, mon_shutdown_cb,
        NULL, NULL,
        mon_conn_msg_event_cb, NULL, &port_ref);

    if (res)
        return res;

    /* send first message to port */
    msg_sect.iov_base = (char *)&subscr;
    msg_sect.iov_len = sizeof(subscr);
}
```



```

tipc_send2name(port_ref,
               &name,
               0 /* domain of 0:own zone */,
               1 /* num_sect */,
               msg_sect);
return OK;
}

```

More examples

Demo programs utilizing the native API can be found at <http://tipc.sf.net>.

In addition, the TIPC source code itself contains a couple of sections that utilize the native API just like an application might:

1) **tipc_cfg_init()** in **net/tipc/config.c**

This file contains the TIPC configuration service (using port name {0,Z.C.N}, which handles messages sent by the tipc-config application. It utilizes a very simple connectionless request-and-reply approach to messaging.

2) **tipc_subscr_start()** in **net/tipc/subscr.c**

This file contains the TIPC topology service (using port name {1,1}), which handles subscription requests from applications and returns subscription events. It demonstrates the correct way to handle connection establishment (both explicit and implied) and tear down (both self-initiated and peer-initiated).

Structures used in the native API

The various structures used for sending and receiving data are important to properly use the native API.

```

/ *
 * VxWorks TIPC emulation of common file socket buffer API
 * /

struct sk_buff {
    struct sk_buff *next;    /* ptr to next buffer in list */
    struct sk_buff *prev;    /* ptr to previous buffer in list */
    M_BLK_ID      mBlkId;    /* ptr to mBlk */
    char          cb[sizeof(struct tipc_skb_cb)]; /* control block area */
    unsigned int   len;
    unsigned char  *data;
    unsigned char  *tail;
};

```

```
/*
 * Declarations used in emulation of common file socket buffer API
 */

struct tipc_skb_cb {
    void *handle;
};

/*
 * VxWorks iovector structure (treat as an array for multiple segments)
 */
struct iovector {
    char *iov_base; /* Base address. */
    size_t iov_len; /* Length. */
};
```

Caveats and Warnings

The native API is a more direct way of using TIPC in kernel applications. There are some usage warnings that one should be aware of when developing an application with this API.

WARNING! The native API is still under development at this time
WARNING! and has not been finalized by the TIPC Working Group. Expect
WARNING! changes in future versions of TIPC.

Accessing a specific port via the native API is not re-entrant. The reason for this is that the header for each message is cached in the port structure and is not locked. Parallel access can result in corrupted header information. Applications should take care to handle this situation.

On SMP systems, when running multiple links between nodes, there is a race condition when the discovery mechanism detects the new node and tries to create a link. A check is done to determine if the node is already known, and then the node structure is allocated and added to the list of known nodes. If the second link detects the same new node and checks for this node after the first link has checked for the node but before the first link has created the node structure, then the duplication of node structures can result in an error. Again, this is something that could only be seen on SMP systems.

For more information and updated notes, please check the release notes as well as the TIPC discussion list. Information about the latter is available at <http://tipc.sourceforge.net> and click on the "Support" tab.

INCLUDE FILES **tipc/tipc.h**

SEE ALSO *Wind River TIPC for VxWorks 6 Programmer's Guide*

B

Socket and Utility Routines

<code>accept()</code>	– accept a request for a connection to a socket.....	148
<code>bind()</code>	– bind an address to a socket	149
<code>close()</code>	– close a socket	150
<code>connect()</code>	– request a connection to a socket	151
<code>getpeername()</code>	– get the port ID of a peer socket.....	152
<code>getsockname()</code>	– get the port ID of a socket	153
<code>getsockopt()</code>	– get the value of an option associated with a socket	154
<code>listen()</code>	– enable a socket to receive connection requests	156
<code>recv()</code>	– receive data from a socket.....	156
<code>recvfrom()</code>	– receive data from a socket.....	158
<code>recvmsg()</code>	– receive data from a socket.....	159
<code>send()</code>	– send a message to a socket.....	161
<code>sendmsg()</code>	– send a message to a socket.....	162
<code>sendto()</code>	– send a message to a socket.....	164
<code>setsockopt()</code>	– set the value of an option associated with a socket	166
<code>shutdown()</code>	– shut down a connection.....	168
<code>socket()</code>	– create a socket	169
<code>tipcConfig()</code>	– the public API for TIPC configuration and management commands.....	170
<code>tipcDataPoolShow()</code>	– display TIPC data-pool statistics.....	171
<code>tipcSysPoolShow()</code>	– display TIPC system-pool statistics	172
<code>tipc_addr()</code>	– combine zone, cluster, and node numbers into a TIPC address	173
<code>tipc_cluster()</code>	– take a TIPC network address and return the cluster number.....	173
<code>tipc_node()</code>	– take a TIPC address and return the node number	174
<code>tipc_zone()</code>	– take a TIPC address and return the zone number	175

accept()

NAME `accept()` – accept a request for a connection to a socket

SYNOPSIS

```
int accept
(
    int          sd,          /* socket descriptor, listening socket */
    struct sockaddr * addr,    /* return param for requester address */
    int *        addrlen     /* return param for address length */
)
```

DESCRIPTION This routine accepts a request for a connection to a socket. It creates a socket for the connection and returns the socket descriptor of the new socket. The routine blocks the caller until a connection is present, unless the socket is marked as non-blocking (see **FIONBIO** under **ioctl()**). The socket *sd* must be a socket that was set as a listening socket with a previous call to **listen()**. The routine is valid only for sockets of type **SOCK_SEQPACKET** and **SOCK_STREAM** (see **socket()**).

Parameters:

sd

The socket descriptor of the listening socket.

addr

A pointer to a **sockaddr** structure for receiving the port ID of the connecting socket. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure. This field can be **NULL** if the port ID is not required by the caller.

addrlen

A pointer to the length, in bytes, of the **sockaddr** structure in the *addr* parameter. Initially, if *addr* is non-**NULL**, it should be set to the size of a **sockaddr_tipc** structure. Upon return, it gives the length of the **sockaddr** structure containing the port ID of the connecting socket.

RETURNS A socket descriptor (a small, non-negative integer) on success, -1 on failure. The socket descriptor that is returned is used to identify the socket in subsequent calls to the socket API.

ERRNO **EINVAL**, **ENOBUFS**, **EOPNOTSUPPORT**, **EWOULDBLOCK**, **EPROTONOTSUP**, **EPROTOPTYPE**, or **ENOMEM**

SEE ALSO **tipc_lib**

bind()

NAME **bind()** – bind an address to a socket

SYNOPSIS

```
STATUS bind
(
    int          sd,          /* socket descriptor */
    struct sockaddr * addr,    /* address to bind to socket */
    int          addrlen      /* length of address */
)
```

DESCRIPTION This routine associates a **sockaddr** structure containing a TIPC port name or port name sequence with the socket identified by the socket descriptor, making it possible for other sockets to connect to or send to it using a predetermined socket address. Typically, the **sockaddr** structure is derived by casting a **sockaddr_tipc** structure as a **sockaddr** structure.

An application can perform multiple bind operations on the same socket. TIPC supports binding multiple port names, port name sequences, or a combination of both to a socket. This is useful if the socket is capable of performing multiple functions within a network. Conversely, a given port name or port name sequence can be bound to multiple sockets within a network. This is useful if more than one socket is capable of performing the associated function within a network.

If the socket that is the target of the bind operation is connected to another socket at the time of the call, the operation fails and generates an error.

The **sockaddr_tipc** structure contains a **scope** field that defines the extend to which the name or name sequence is published. The valid entries are **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE** as appropriate.

A name or name sequence that was associated with a socket using **bind()** can be unbound by calling **bind()** again using the negative of the scope value used originally (eg. use **-TIPC_CLUSTER_SCOPE** instead of **TIPC_CLUSTER_SCOPE**). To unbind all names and name sequences at once, pass in a socket address of length zero.

NOTE: When a socket is created, TIPC automatically assigns it a port ID. An application can use the port ID as the socket's address without having to bind the port ID to the socket through a **bind()** operation.

Parameters:

sd

The socket descriptor of the socket to bind an address to.

addr

A pointer to a **sockaddr** structure containing the port name or port name sequence to bind to the socket. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure.

addrlen

The length, in bytes, of the **sockaddr** structure in the *addr* parameter.

RETURNS	OK on success, or ERROR if the socket does not exist, the address is invalid, or the socket is already connected.
ERRNO	N/A
SEE ALSO	tipc_lib

close()

NAME **close()** – close a socket

SYNOPSIS

```
STATUS close
(
    int sd          /* socket descriptor */
)
```

DESCRIPTION This routine closes a socket, and frees all resources associated with it.

If messages sent in a connectionless and reliable manner (**SOCK_RDM** or **SOCK_DGRAM** with the **TIPC_DEST_DROPPABLE** option not set (see **setsockopt()**)) are still in the socket's receive queue, each message is rejected and the first 1024 bytes of each message are returned to the sender. Any other connectionless messages still in the socket's receive queue are discarded.

If messages sent in a connection-oriented and reliable manner (**SOCK_SEQPACKET** or **SOCK_STREAM** with the **TIPC_DEST_DROPPABLE** option not set (see **setsockopt()**)) are still in the socket's receive queue, the first message is rejected and its first 1024 bytes are returned to the sender; the remaining messages are discarded. If the socket is connected to another socket at the time it is closed, the connection is terminated.

NOTE: It is advisable to perform a **shutdown()** on a connection-oriented socket that was connected to a peer before calling **close()** so that the peer can easily distinguish between a connection that was intentionally and properly terminated versus one that was abnormally terminated. See **recv()** and **recvmsg()**.

NOTE: VxWorks does not employ reference counting to ensure that there are no other users of the socket at the time of closure, so applications should not close a socket if it is being used by another thread of control.

Parameter:

sd

Socket descriptor of the socket to close.

RETURNS **OK** on success, **ERROR** otherwise.

ERRNO **EINVAL**

SEE ALSO **tipc_lib**

connect()

NAME **connect()** – request a connection to a socket

SYNOPSIS

```
STATUS connect
(
    int          sd,          /* socket descriptor, requesting socket */
    struct sockaddr * addr,   /* address of socket to connect to */
    int          addrlen /* length of the address, in bytes */
)
```

DESCRIPTION This routine requests a connection to another socket as defined in the *addr* structure. It is valid only with source and destination sockets of type **SOCK_SEQPACKET** and **SOCK_STREAM** (see **socket()**). The socket requesting the connection cannot have a name or name sequence bound to it.

The **connect()** routine blocks until one of the following occurs:

- The destination system accepts the connection.
- The socket's connect time limit is reached (see **CONN_ACK_TIMEOUT** under **setsockopt()**).
- The specified destination cannot be located or is located but then ceases to exist.
- The connection attempt fails or is invalid (for example, because the socket has a port name or port name sequence bound to it).

Parameters:

sd

A socket descriptor identifying the socket making a connection request.

name

A pointer to a **sockaddr** structure containing the port ID or port name of the destination socket. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure.

namelen

The length, in bytes, of the **sockaddr** structure in the *name* parameter.

RETURNS **OK** on success, **ERROR** otherwise.

ERRNO EOPNOTSUPP, EINPROGRESS, EINVAL, ETIMEDOUT, EALREADY, EDESTADDRREQ, or EISCONN

SEE ALSO `tipc_lib`

getpeername()

NAME `getpeername()` – get the port ID of a peer socket

SYNOPSIS

```
STATUS getpeername
(
    int          sd,          /* the socket requesting the peer name */
    struct sockaddr * addr,   /* return param for the port ID */
    int *        addrlen     /* return param for address length */
)
```

DESCRIPTION This routine gets the port ID of the peer socket that is connected to the specified socket. It is valid only for a socket of type **SOCK_SEQPACKET** or **SOCK_STREAM** (see `socket()`).

Parameters:

sd

The socket descriptor of the socket requesting the peer's port ID.

addr

A pointer to a **sockaddr** structure for returning the port ID of the peer socket. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure.

addrlen

The length, in bytes, of the **sockaddr** structure in the *addr* parameter. Initially, it should be set to the size of a **sockaddr_tipc** structure. Upon return, it gives the length of the **sockaddr** structure containing the peer socket's port ID.

RETURNS **OK** on success, **ERROR** otherwise.

ERRNO EFAULT, EBADF, or ENOTSUP

SEE ALSO `tipc_lib`

getsockname()

NAME `getsockname()` – get the port ID of a socket

SYNOPSIS

```
STATUS getsockname
(
    int                sd,          /* socket descriptor */
    struct sockaddr *  addr,        /* return parameter for the port ID */
    int *              addrlen     /* return parameter for address length */
)
```

DESCRIPTION This routine returns the port ID of a socket.

NOTE: This routine simply returns the port ID of a socket and the use of **name** may be misleading as there is no relation to a TIPC port name or port name sequence that may be associated with a port.

Parameters:

sd

The socket descriptor of the socket.

addr

A pointer to a **sockaddr** structure for returning the port ID of the socket. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure.

addrlen

The length, in bytes, of the **sockaddr** structure in the *addr* parameter. Initially, it should be set to the size of a **sockaddr_tipc** structure. Upon return, it gives the length of the **sockaddr** structure containing the socket's port ID.

RETURNS **OK** on success, **ERROR** otherwise.

ERRNO **EINVAL**, **EFAULT**, **EBADE**, or **ENOTSUP**

SEE ALSO `tipc_lib`

getsockopt()

NAME `getsockopt()` – get the value of an option associated with a socket

SYNOPSIS

```
STATUS getsockopt
(
    int     sd,           /* socket descriptor */
    int     level,        /* protocol level, always SOL_TIPC */
    int     optname,       /* a TIPC-specific option name */
    char *  optval,        /* return parameter for the value of the option */
    int *   optlen         /* return parameter for the length of optval */
)
```

DESCRIPTION This routine gets the current value of a TIPC-specific option associated with a socket.

Parameters:

sd

The socket descriptor of the target socket.

level

The protocol level of the option, always **SOL_TIPC**. Wind River TIPC does not support **SOL_SOCKET**-level options.

NOTE: For **SOCK_STREAM** sockets only, returns 0 length for all **IPPROTO_TCP** options (to ease compatibility and conversion to TIPC).

optname

The name of the option for which a value is to be retrieved. The following options are available:

TIPC_IMPORTANCE

The importance of messages sent through the socket. The lower the importance, the more likely the message is to be discarded due to congestion in the TIPC network. If messages are sent in a reliable manner, using a lower importance can result in delays, since messages may need to be resent. If messages are sent in an unreliable manner, this can result in lost messages.

The following values can be returned:

TIPC_LOW_IMPORTANCE

TIPC_MEDIUM_IMPORTANCE

TIPC_HIGH_IMPORTANCE

TIPC_CRITICAL_IMPORTANCE

The default value is **TIPC_LOW_IMPORTANCE**.

TIPC_SRC_DROPPABLE

This option governs the handling of messages sent by the socket if link congestion occurs. If enabled, the message is discarded; otherwise the system queues the message for later transmission.

By default, this option is disabled for **SOCK_SEQPACKET**, **SOCK_STREAM**, and **SOCK_RDM** socket types (resulting in "reliable" data transfer), and enabled for **SOCK_DGRAM** (resulting in "unreliable" data transfer).

TIPC_DEST_DROPPABLE

This option governs the handling of messages sent by the socket if the message cannot be delivered to its destination, either because the receiver is congested or because the specified receiver does not exist. If enabled, the message is discarded; otherwise the message is returned to the sender.

By default, this option is disabled for **SOCK_SEQPACKET** and **SOCK_STREAM** socket types, and enabled for **SOCK_RDM** and **SOCK_DGRAM**. This arrangement ensures proper teardown of failed connections when connection-oriented data transfer is used, without increasing the complexity of connectionless data transfer.

CONN_ACK_TIMEOUT

The number of milliseconds that **connect()** waits for a connection to be established before abandoning the connection attempt.

optval

A pointer to a buffer for returning the value of the specified option. Although **optval** is passed in as **char ***, the option value whose address gets passed in is an integer, whose address needs to be cast to a pointer to **char**.

optlen

A pointer to the length, in bytes, of the option value to return.

RETURNS **OK** on success, **ERROR** otherwise.

ERRNO **EINVAL** or **ENOPROTOOPT**

SEE ALSO **tipc_lib**

listen()

NAME	listen() – enable a socket to receive connection requests
SYNOPSIS	<pre>STATUS listen (int sd, /* socket descriptor */ int backlog /* connect-request max queue length--ignored by TIPC */)</pre>
DESCRIPTION	<p>This routine enables a socket to receive (listen for) connection requests. It is valid only with a sockets of type SOCK_SEQPACKET or SOCK_STREAM (see socket()).</p> <p>Parameters:</p> <p><i>sd</i></p> <p>The socket descriptor of the socket that is to listen for connection requests.</p> <p><i>backlog</i></p> <p>TIPC ignores the value assigned to this parameter, which otherwise specifies the maximum length of the queue waiting for connections.</p>
RETURNS	OK on success, ERROR otherwise.
ERRNO	N/A
SEE ALSO	tipc_lib

recv()

NAME	recv() – receive data from a socket
SYNOPSIS	<pre>int recv (int sd, /* socket descriptor */ char * buf, /* pointer to a buffer for receiving data */ int bufLen, /* length of buffer */ int flags /* flags to underlying protocols */)</pre>
DESCRIPTION	<p>This routine allows a socket to receive a message. The recv() routine can be used with both connectionless (SOCK_DGRAM, SOCK_RDM) and connection-oriented (SOCK_SEQPACKET, SOCK_STREAM) sockets.</p>

TIPC allows a socket to receive messages sent by another socket. In order to receive returned messages (for example, because the destination socket was closed or the destination address does not exist) the **recvmsg()** function should be used.

For connectionless sockets, a return value of 0 indicates the return of an undelivered data message that was originally sent by this socket.

For connection-oriented sockets, a return value of 0 indicates that the connection was terminated by the peer issuing a **shutdown()**. A return value of -1 indicates that the connection was terminated for some other reason.

TIPC supports the **MSG_PEEK** and **MSG_DONTWAIT** flags when receiving, as well as the **MSG_WAITALL** flag when receiving on a **SOCK_STREAM** socket; all other flags are ignored.

Parameters:

sd

The socket that receives the data.

buf

A pointer to a buffer for receiving data. For socket types other than **SOCK_STREAM**, if the buffer is not large enough to hold the message, the message is truncated and the excess data is discarded.

bufLen

The size of the buffer, in bytes.

flags

The following flags are available. They can be **OR**-ed.

MSG_PEEK

Allows the application to receive a message without removing it from the socket's receive queue.

MSG_DONTWAIT

Prevents **recv()** from blocking if the socket's receive queue is currently empty. The same effect can be achieved using the **ioctl()** call with **FIONBIO**.

MSG_WAITALL

For **SOCK_STREAM** sockets only, causes the **recv()** to block until all data specified has been received.

RETURNS

The number of bytes received, 0 for a connection that terminated normally or a returned message was detected, or **ERROR** if the call fails.

ERRNO

EINVAL, **EOPNOTSUPP**, **EWOULDBLOCK**, **ECONNRESET**, or **ENOTCONN**

SEE ALSO

tipc_lib, **recvmsg()**

recvfrom()

NAME `recvfrom()` – receive data from a socket

SYNOPSIS

```
int recvfrom
(
    int          sd,          /* socket descriptor of receiving socket */
    char *       buf,         /* pointer to buffer for receiving data */
    int          bufLen,      /* length of buffer */
    int          flags,       /* flags to underlying protocols */
    struct sockaddr * from,    /* return parameter for sender's address */
    int *        fromLen      /* return param for length of address */
)
```

DESCRIPTION This routine receives a message from either a connectionless (**SOCK_DGRAM**, **SOCK_RDM**) socket or a connection-oriented (**SOCK_SEQPACKET**, **SOCK_STREAM**) socket.

TIPC allows a socket to receive messages sent by another socket. In order to receive returned messages (for example, because the destination socket was closed or the destination address does not exist) the **recvmsg()** function should be used.

TIPC supports the **MSG_PEEK** and **MSG_DONTWAIT** flags when receiving, as well as the **MSG_WAITALL** flag when receiving on a **SOCK_STREAM** socket; all other flags are ignored.

Parameters:

sd

The socket that receives the data.

buf

A pointer to a buffer for receiving data.

For socket types other than **SOCK_STREAM**, if the buffer is not large enough to hold the message, the message is truncated and the excess data is discarded.

bufLen

The size of the buffer, in bytes.

flags

The following flags are available. They can be OR-ed.

MSG_PEEK

Allows the application to receive some or all of a message without removing it from the socket's receive queue.

MSG_DONTWAIT

Prevents **recvfrom()** from blocking if the socket's receive queue is currently empty. The same effect can be achieved on the socket using the **ioctl()** call with **FIONBIO**.

MSG_WAITALL

For **SOCK_STREAM** sockets only, causes the **recv()** to block until all data specified has been received.

from

A return parameter that points to a **sockaddr** structure for holding the port identifier of the sender. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure.

fromLen

The length, in bytes, of the **sockaddr** structure in the *from* parameter. Initially, it should be set to the size of a **sockaddr_tipc** structure. Upon return, it contains the size of the specific instance of the **sockaddr** structure in the **from** parameter.

RETURNS

The number of bytes received, 0 for a connection that terminated normally or a returned message was detected, or **ERROR** if the call fails.

ERRNO

EINVAL, **EOPNOTSUPP**, or **EWOULDBLOCK**

SEE ALSO

tipc_lib, **recvmsg()**

recvmsg()

NAME

recvmsg() – receive data from a socket

SYNOPSIS

```
int recvmsg
(
    int          sd,           /* socket descriptor */
    struct msghdr * msg,       /* pointer to a message structure receiving
                               * both source address and incoming data */
    int          flags         /* flags to underlying protocols */
)
```

DESCRIPTION

This routine allows a socket to receive a message as well as ancillary data and the port ID of the sender, if requested. The **recvmsg()** routine can be used with both connectionless (**SOCK_DGRAM**, **SOCK_RDM**) and connection-oriented (**SOCK_SEQPACKET**, **SOCK_STREAM**) sockets.

For connectionless sockets, a return value of 0 indicates the return of an undelivered data message that was originally sent by this socket.

For connection-oriented sockets, a return value of 0 or -1 indicates connection termination. The exact return value upon connection termination is influenced by the "msg_control" field of "msg". If "msg_control" is **NULL**, a return value of 0 indicates that the connection was terminated by the peer using **shutdown()**; connection termination by any other means causes a return value of -1. If "msg_control" is non-**NULL**, a return value of 0 is always used; the application must examine the **TIPC_ERRINFO** object to determine if the connection was explicitly terminated by the peer. (POSIX non-conformity)

The port ID of the message sender is captured in the "msg_name" field of "msg" (if non-**NULL**) and ancillary data relating to the message is captured in the "msg_control" field of "msg" (if non-**NULL**). The data portion of the message is stored in the "msg_iov" field.

The following ancillary data objects may be captured:

- 1) **TIPC_ERRINFO** - The TIPC error code associated with a returned data message or a connection termination message, and the length of the returned data. (8 bytes: error code + data length)
- 2) **TIPC_RETDATA** - The contents of a returned data message, up to a maximum of 1024 bytes.
- 3) **TIPC_DESTNAME** - The TIPC name or name sequence that was specified by the sender of the message. (12 bytes: type + lower instance + upper instance; the latter two values are the same for a TIPC name, but may differ for a name sequence)

Each of these objects is only created where relevant. For example, receipt of a normal data message never creates the **TIPC_ERRINFO** and **TIPC_RETDATA** objects, and only creates the **TIPC_DESTNAME** object if the message was sent using a TIPC name or name sequence as the destination rather than a TIPC port ID. Those objects that are created will always appear in the relative order shown above.

If ancillary data object capture is requested (i.e. "msg->msg_control" is non-**NULL**) but insufficient space is provided, the **MSG_CTRUNC** flag is set to indicate that one or more available objects were not captured.

When used with connection-oriented sockets, **TIPC_DESTNAME** is captured for each data message received by the socket if the connection was established using a TIPC name or name sequence as the destination address. Note: There is currently no way for the destination socket to capture **TIPC_DESTNAME** following **accept()** until the originator sends a data message.

TIPC supports the **MSG_PEEK** and **MSG_DONTWAIT** flags when receiving, as well as the **MSG_WAITALL** flag when receiving on a **SOCK_STREAM** socket; all other flags are ignored.

Parameters:

sd

The socket that receives the data.

msg

A pointer to a struct **msg_hdr** for receiving data, ancillary data, and sender information.

The `msg_name` and `msg_namelen` fields define space to store the sender's port ID.

The `msg_control` and `msg_controllen` fields define space to store ancillary data.

The `msg_iov` and `msg_iovlen` fields define space for message data. NOTE: `recvmsg()` supports only a single `iov`.

flags

The following flags are available. They can be **OR**-ed.

MSG_PEEK

Allows the application to receive a message without removing it from the socket's receive queue.

MSG_DONTWAIT

Prevents `recvmsg()` from blocking if the socket's receive queue is currently empty. The same effect can be achieved using the `ioctl()` call with **FIONBIO**.

MSG_WAITALL

For **SOCK_STREAM** sockets only, causes the `recv()` to block until all data specified has been received.

RETURNS	The number of bytes received, 0 for a connection that terminated normally and/or a returned message was detected, or ERROR if the call fails.
ERRNO	EINVAL , EOPNOTSUPP , EWOULDBLOCK , ECONNRESET , or ENOTCONN
SEE ALSO	<code>tipc_lib</code> , <code>recv()</code>

send()

NAME	<code>send()</code> – send a message to a socket
SYNOPSIS	<pre>int send (int sd, /* socket descriptor of sending socket */ const char * buf, /* pointer to a buffer for the message */ int bufLen, /* length of the buffer */ int flags /* always 0; not used by Wind River TIPC */)</pre>
DESCRIPTION	<p>This routine sends a message over a previously established connection. It is valid only for sockets of type SOCK_SEQPACKET or SOCK_STREAM.</p> <p>The <code>send()</code> routine should not be used until a connection has been fully established using either explicit or implicit handshaking.</p>

TIPC supports the `MSG_DONTWAIT` flag when sending; all other flags are ignored.

Parameters:

sd
Socket descriptor of the socket sending the message.

buf
A pointer to a buffer for the message.

bufLen
A positive value specifying the length of the buffer.

flags
The following flag is available.

MSG_DONTWAIT
Prevents `send()` from blocking if there is congestion on the link. The same effect can be achieved using the `ioctl()` call with **FIONBIO**.

RETURNS	The number of bytes sent, or ERROR if the call fails.
ERRNO	EINVAL , EWOULDBLOCK , EPIPE , ENOTCONN , EOPNOTSUPP , or EISCONN .
SEE ALSO	<code>tipc_lib</code>

sendmsg()

NAME `sendmsg()` – send a message to a socket

SYNOPSIS

```
int sendmsg
(
    int          sd,          /* socket descriptor of sending socket */
    struct msghdr * msg,      /* pointer to a message structure sending
                               * both source address and outgoing data */
    int          flags        /* flags to underlying protocols */
)
```

DESCRIPTION This routine sends a message to a socket. TIPC allows the routine to be used with both connectionless (**SOCK_DGRAM**, **SOCK_RDM**) and connection-oriented (**SOCK_SEQPACKET**, **SOCK_STREAM**) sockets.

Connectionless Socket: When `sendmsg()` is used with a connectionless socket, it transmits a message to a one or more destination sockets, which can be specified by port name, port identity, or, in the case of a multicast message, by port name sequence.

If the destination is denoted by a TIPC name or a port ID the message is unicast to a single port; if the destination is denoted by a TIPC name sequence the message is multicast to all ports having a TIPC name or name sequence that overlaps the destination name sequence.

Connection-Oriented Socket: When **sendmsg()** is used with a connection-oriented socket, a connection initiation and message send is achieved in a single operation, rather than performing separate **connect()** and **send()** operations; this technique is known as an implied connect.

A connection to the peer is not fully established until the application successfully uses **recv()**, **recvfrom()**, or **recvmsg()** to receive the message sent back by the peer socket. While waiting for the connection, the application cannot use **sendmsg()** again to send a message to the same destination. Once the connection is established, the application can also transmit messages to the peer using **send()**.

The implied-connect technique is most suited to situations where a client needs to send a single request to a server and receive a single reply (which may consist of multiple messages). It is faster than using the **connect()** and **send()** routines in sequence and, because it is connection-oriented, it still guarantees a correlation between a request and the response to it.

TIPC supports the **MSG_DONTWAIT** flag when sending; all other flags are ignored.

TIPC does not currently support the use of ancillary data with **sendmsg()**.

Parameters:

sd

Socket descriptor of the socket sending the message.

msg

A pointer to a struct **msg_hdr** for sending data and destination information.

The **msg_name** and **msg_namelen** fields point to the destination information. If **msg_name** is non-NULL, then **msg_namelen** must specify a size at least as large as the size of the **sockaddr_tipc** structure.

The **msg_control** and **msg_controllen** fields are ignored since ancillary data is not supported for **sendmsg()**.

The **msg_iov** and **msg_iovlen** fields define space for message data.

flags

The following flag is available.

MSG_DONTWAIT

Prevents **sendmsg()** from blocking if there is congestion on the link. The same effect can be achieved using the **ioctl()** call with **FIONBIO**.

RETURNS	The number of bytes sent, or ERROR if the call fails.
ERRNO	EINVAL , EWOULDBLOCK , EPIPE , ENOTCONN , EOPNOTSUPP , or EISCONN .
SEE ALSO	tipc_lib

sendto()

NAME **sendto()** – send a message to a socket

SYNOPSIS

```
int sendto
(
    int          sd,          /* socket descriptor of sending socket */
    char *       buf,         /* pointer to a buffer for the message */
    int          bufLen,      /* the length of the buffer */
    int          flags,       /* flags to underlying protocols */
    struct sockaddr * to,     /* address of the destination socket */
    int          tolen        /* length of the address */
)
```

DESCRIPTION This routine sends a message to a socket. TIPC allows the routine to be used with both connectionless (**SOCK_DGRAM**, **SOCK_RDM**) and connection-oriented (**SOCK_SEQPACKET**, **SOCK_STREAM**) sockets.

Connectionless Socket: When **sendto()** is used with a connectionless socket, it transmits a message to a one or more destination sockets, which can be specified by port name, port identity, or, in the case of a multicast message, by port name sequence.

If the destination is denoted by a TIPC name or a port ID the message is unicast to a single port; if the destination is denoted by a TIPC name sequence the message is multicast to all ports having a TIPC name or name sequence that overlaps the destination name sequence.

Connection-Oriented Socket: When **sendto()** is used with a connection-oriented socket, a connection initiation and message send is achieved in a single operation, rather than performing separate **connect()** and **send()** operations; this technique is known as an implied connect.

A connection to the peer is not fully established until the application successfully uses **recv()**, **recvfrom()**, or **recvmsg()** to receive the message sent back by the peer socket. While waiting for the connection, the application cannot use **sendto()** again to send a message to the same destination. Once the connection is established, the application transmits messages to the peer using **send()**, not **sendto()**.

The implied-connect technique is most suited to situations where a client needs to send a single request to a server and receive a single reply (which may consist of multiple messages). It is faster than using the **connect()** and **send()** routines in sequence and,

because it is connection-oriented, it still guarantees a correlation between a request and the response to it.

Parameters:

sd
Socket descriptor of the socket sending the message.

buf
A pointer to a buffer for the message.

bufLen
A positive value specifying the length of the buffer.

flags
The following flag is available.

MSG_DONTWAIT

Prevents **sendto()** from blocking if there is congestion on the link. The same effect can be achieved using the **ioctl()** call with **FIONBIO**.

to
A pointer to a **sockaddr** structure containing the port identifier, port name, or port name sequence of the destination. Typically, this parameter points to a **sockaddr_tipc** structure that is cast as a **sockaddr** structure. If this parameter points to a TIPC port name, then the **addr.name.domain** field indicates search domain used during the name lookup process. (In contrast, if this parameter is set to a TIPC name sequence the message is multicast to all ports having a TIPC name or name sequence that overlaps the destination name sequence. Additionally, if this parameter points to a TIPC port ID, no name lookup occurs.) The "scope" field of this parameter is always ignored when sending.

toLen
The length of the **sockaddr** structure in the **to** parameter.

RETURNS The number of bytes sent, or **ERROR** if the call fails.

ERRNO **EINVAL**, **EWOULDBLOCK**, **EPIPE**, **ENOTCONN**, **EOPNOTSUPP**, or **EISCONN**.

SEE ALSO **tipc_lib**

setsockopt()

NAME **setsockopt()** – set the value of an option associated with a socket

SYNOPSIS

```
STATUS setsockopt
(
    int    sd,           /* socket descriptor of the socket */
    int    level,        /* protocol level, always SOL_TIPC */
    int    optname,      /* a TIPC-specific option name */
    char * optval,       /* the value to assign to the option */
    int    optlen        /* length of the option value */
)
```

DESCRIPTION This routine sets the value of a TIPC-specific option associated with a socket.

Parameters:

sd

The socket descriptor of the socket.

level

The protocol level of the option, always **SOL_TIPC**. Wind River TIPC does not support **SOL_SOCKET**-level options.

For **SOCK_STREAM** sockets only, the value of **IPPROTO_TCP** is allowed, but is ignored.

optname

The name of the option to set. The following options are available:

IMPORTANCE_OPTION

Sets the importance of messages sent through the target socket. The lower the importance, the more likely the message is to be discarded due to congestion in the TIPC network. If messages are sent in a reliable manner, this can result in delays, since messages may need to be resent. If messages are sent in an unreliable manner, this can result in lost messages.

This option can be set to the following values:

TIPC_LOW_IMPORTANCE

TIPC_MEDIUM_IMPORTANCE

TIPC_HIGH_IMPORTANCE

TIPC_CRITICAL_IMPORTANCE

The default value is **TIPC_LOW_IMPORTANCE**.

TIPC_SRC_DROPPABLE

This option governs the handling of messages sent by the socket if link congestion occurs. If enabled, the message is discarded; otherwise the system queues the message for later transmission.

The default value of **TIPC_SRC_DROPPABLE** for **SOCK_SEQPACKET**, **SOCK_STREAM**, and **SOCK_RDM** socket types is 0 (disabled) resulting in "reliable" data transfer. The default value for **SOCK_DGRAM** is non-zero (enabled) for "unreliable" data transfer.

TIPC_DEST_DROPPABLE

This option governs the handling of messages sent by the socket if the message cannot be delivered to its destination, either because the receiver is congested or because the specified receiver does not exist. If enabled, the message is discarded; otherwise the message is returned to the sender.

The default value of **TIPC_DEST_DROPPABLE** for **SOCK_SEQPACKET** and **SOCK_STREAM** socket types is 0 (disabled). The default value for **SOCK_RDM** and **SOCK_DGRAM** is non-zero (enabled). This arrangement ensures proper teardown of failed connections when connection-oriented data transfer is used, without increasing the complexity of connectionless data transfer.

CONN_ACK_TIMEOUT

Specifies the number of milliseconds that **connect()** waits for a connection to be established before abandoning the connection attempt.

The default value for **CONN_ACK_TIMEOUT** is 8 seconds.

optval

A pointer to the value to set for the specified option. Although *optval* is a pointer to **char**, all the underlying option values are integers, therefore the option value must be cast as a pointer to **char**.

optlen

The length of the option value in bytes.

RETURNS **OK** on success, **ERROR** otherwise.

ERRNO **EINVAL** or **ENOPROTOOPT**

SEE ALSO **tipc_lib**

shutdown()

NAME	shutdown() – shut down a connection
SYNOPSIS	<pre>STATUS shutdown (int sd, /* identifies the socket to shut down */ int how /* function code */)</pre>
DESCRIPTION	<p>Shuts down socket send and receive operations on a connection-oriented socket. The socket's peer is notified that the connection was deliberately terminated by the application (by means of the TIPC_CONN_SHUTDOWN error code), rather than as the result of an error.</p> <p>TIPC does not support partial shutdown of a connection; attempting to shut down either send or receive operations always shuts down both.</p> <p>Applications should normally call shutdown() to terminate a connection before calling close().</p> <p>A socket that has been shutdown() cannot be re-used for a new connection; this prevents any "stale" incoming messages from an earlier connection from interfering with the new connection.</p> <p>Parameters:</p> <p><i>sd</i> The socket descriptor of the socket to shut down.</p> <p><i>how</i> TIPC will only accept a value of SHUT_RDWR for the <i>how</i> parameter. Any other passed value will result in an error (EINVAL).</p> <p>TIPC only supports a complete shut down of the socket.</p>
RETURNS	OK on success, ERROR otherwise
ERRNO	EBADF , ENOTSUP , EINVAL , or ENOTCONN
SEE ALSO	tipc_lib

socket()

NAME socket() – create a socket

SYNOPSIS

```
int socket
(
    int domain,          /* address family: AF_TIPC */
    int type,            /* SOCK_SEQPACKET, SOCK_STREAM, SOCK_DGRAM, or
SOCK_RDM */
    int protocol         /* socket protocol, always 0 */
)
```

DESCRIPTION This routine opens a socket and returns a socket descriptor. The socket descriptor is passed to the other socket routines to identify the socket. The socket descriptor is a standard I/O system file descriptor (fd) and can be used with the **close()**, **read()**, **write()**, and **ioctl()** routines.

Parameters:

domain
The addressing protocol to use, **AF_TIPC**.

type
One of the following types of socket:

Type	Description
SOCK_SEQPACKET	Transfer messages in a reliable, connection-oriented manner.
SOCK_STREAM	Transfer byte streams in a reliable, connection-oriented manner.
SOCK_DGRAM	Transfer messages in an unreliable, connectionless manner.
SOCK_RDM	Transfer messages in a reliable, connectionless manner.

protocol
The socket protocol, always 0.

RETURNS A socket descriptor (a small, non-negative integer) on success, -1 on failure. The socket descriptor is used to identify the socket in subsequent calls to the socket API.

ERRNO ENOBUFS, EPROTONOSUPPORT, or ENOMEM

SEE ALSO tipc_lib

B

tipcConfig()

NAME `tipcConfig()` – the public API for TIPC configuration and management commands

SYNOPSIS

```
void tipcConfig
(
    char * str /* command string */
)
```

DESCRIPTION This routine displays or sets various parameters and statistics in the TIPC module. The routine is included as a Show routine and is intended to allow the configuration and management of TIPC in a network. Many commands are privileged and will only work on the local node, but some commands can be executed on remote nodes as well.

Example:

```
-> tipcConfig
```

Usage:

```
tipcConfig option [option ...]
```

```
valid options:
-v                               Toggle Verbose mode
-i                               Toggle Interactive set

operations
-dest [=<addr>]                 Get/set Command destination node
-addr [=<addr>]                 Get/set node address
-netid[=<value>]                 Get/set network id
-mng [=enable|disable]          Get/set remote management
-nt [= [<depth>,<type>[,<low>[,<up>]]] Get name table
    where <depth> = types|names|ports|all

-p                               Get port info
-m                               Get media
-b [=<pattern>]                 Get bearers
-be =<bname>[/<domain>[/<priority>]] Enable bearer
-bd =<bname>|<pattern>           Disable bearer
-n [=<addr>]                     Get nodes in domain
-l [=<addr>|<pattern>]           Get links for domain
-ls [=<linkname>|<pattern>]       Get link statistics
-lsr =<linkname>|<pattern>         Reset link statistics
-lp =<linkname>|<pattern>/<value> Set link priority
-lt =<linkname>|<pattern>/<value> Set link tolerance
-lw =<linkname>|<pattern>/<value> Set link window
-max_ports [=<value>]           Get/set max number of ports
-max_nodes [=<value>]           Get/set max nodes in own cluster
-max_clusters [=<value>]         Get/set max clusters in own zone
-max_zones [=<value>]           Get/set max zones in own network
-max_remotes [=<value>]         Get/set max non-cluster

neighbors
-max_publ [=<value>]             Get/set max publications
-max_subscr [=<value>]           Get/set max subscriptions
-log [=<size>]                   Dump/resize log
-s                               Get TIPC status info
-V                               Program version
```

```
-help                                This usage list

  where <pattern> is an optional search string starting with '?'
->
```

RETURNS N/A

ERRNO N/A

SEE ALSO **tipc_config_show**

tipcDataPoolShow()

NAME **tipcDataPoolShow()** – display TIPC data-pool statistics

SYNOPSIS void tipcDataPoolShow (void)

DESCRIPTION This routine displays statistics on the allocation and availability of clusters in the TIPC data pool. The TIPC data pool is used for the transfer of data packets between TIPC sockets.

Example:

```
-> tipcDataPoolShow
type          number
-----
FREE         :    471
DATA         :      1
TOTAL        :    472
number of mbufs: 472
number of times failed to find space: 0
number of times waited for space: 0
number of times drained protocols for space: 0
```

CLUSTER POOL TABLE						
size	clusters	free	usage	minsize	maxsize	empty
64	120	119	4	64	64	0
128	200	200	0	0	0	0
256	40	40	0	0	0	0
512	40	40	0	0	0	0
1024	50	50	0	0	0	0
2048	20	20	0	0	0	0
4096	2	2	0	0	0	0

RETURNS N/A

ERRNO N/A

SEE ALSO `tipc_config_show`, `netPoolShow()`

tipcSysPoolShow()

NAME `tipcSysPoolShow()` – display TIPC system-pool statistics

SYNOPSIS `void tipcSysPoolShow (void)`

DESCRIPTION This routine displays statistics on the allocation and availability of clusters in the TIPC system pool. The TIPC system pool is used by TIPC sockets and their protocol control blocks.

Example:

```
-> tipcSysPoolShow
type          number
-----
FREE         :    401
TOTAL        :    401
number of mbufs: 401
number of times failed to find space: 0
number of times waited for space: 0
number of times drained protocols for space: 0
```

CLUSTER POOL TABLE

size	clusters	free	usage	minsize	maxsize	empty
16	200	200	0	0	0	0
192	200	200	0	0	0	0
528	200	200	0	0	0	0

Note that parentheses are not required when the routine is invoked from the command line.

RETURNS N/A

ERRNO N/A

SEE ALSO `tipc_config_show`, `netPoolShow()`

tipc_addr()

NAME	tipc_addr() – combine zone, cluster, and node numbers into a TIPC address
SYNOPSIS	<pre>__u32 tipc_addr (unsigned int zone, /* zone number */ unsigned int cluster, /* cluster number */ unsigned int node /* node number */)</pre>
DESCRIPTION	<p>This routine takes individual zone, cluster, and node numbers and combines them into a 32-bit TIPC network address.</p> <p>Parameters:</p> <p><i>zone</i> The zone number.</p> <p><i>cluster</i> The cluster number.</p> <p><i>node</i> The node number.</p>
RETURNS	32-bit network address.
ERRNO	N/A
SEE ALSO	tipc_lib

tipc_cluster()

NAME	tipc_cluster() – take a TIPC network address and return the cluster number
SYNOPSIS	<pre>unsigned int tipc_cluster (__u32 addr /* TIPC network address */)</pre>
DESCRIPTION	<p>This routine takes a 32-bit TIPC network address and returns the cluster number contained in the address.</p>

Parameters:
addr
TIPC network address.

RETURNS Cluster number.

ERRNO N/A

SEE ALSO **tipc_lib**

tipc_node()

NAME **tipc_node()** – take a TIPC address and return the node number

SYNOPSIS

```
unsigned int tipc_node
(
    __u32 addr          /* TIPC network address */
)
```

DESCRIPTION This routine takes a 32-bit TIPC network address and returns the node number contained in the address.

Parameters:

addr
TIPC network address.

RETURNS Node number.

ERRNO N/A

SEE ALSO **tipc_lib**

tipc_zone()

NAME	tipc_zone() – take a TIPC address and return the zone number
SYNOPSIS	<pre>unsigned int tipc_zone (__u32 addr /* TIPC network address */)</pre>
DESCRIPTION	<p>This routine takes a 32-bit TIPC network address and returns the zone number contained in the address.</p> <p>Parameters:</p> <p><i>addr</i> TIPC network address.</p>
RETURNS	Zone number.
ERRNO	N/A
SEE ALSO	tipc_lib

C

TIPC Native Routines

<code>tipc_attach()</code>	– Register a TIPC user	179
<code>tipc_connect2port()</code>	– Associate a TIPC port with its peer	180
<code>tipc_createport()</code>	– Create a TIPC port	181
<code>tipc_deleteport()</code>	– Delete a TIPC port	187
<code>tipc_detach()</code>	– Unregister a TIPC user	188
<code>tipc_disconnect()</code>	– Disassociate a TIPC port with its peer	188
<code>tipc_forward2name()</code>	– Forward a message to the named port	189
<code>tipc_forward2port()</code>	– Forward a message to a port	190
<code>tipc_forward_buf2name()</code>	– Forward a buffer to the named port	191
<code>tipc_forward_buf2port()</code>	– Forward a buffer to a port	192
<code>tipc_get_addr()</code>	– Get the network address for this node	194
<code>tipc_get_mode()</code>	– Get operating mode of TIPC	194
<code>tipc_isconnected()</code>	– Determine if a TIPC port is connected	195
<code>tipc_ispublished()</code>	– Determine if a TIPC name exists	195
<code>tipc_multicast()</code>	– Multicast data to a set of named TIPC ports	196
<code>tipc_ownidentity()</code>	– Get port ID of TIPC port	197
<code>tipc_peer()</code>	– Get the port ID of a TIPC port's peer	198
<code>tipc_portimportance()</code>	– Get importance of TIPC port messages	198
<code>tipc_portunreliable()</code>	– Get reliability of TIPC port messages	199
<code>tipc_portunreturnable()</code>	– Get returnability of TIPC port messages	200
<code>tipc_publish()</code>	– Add a name or name sequence to a TIPC port	200
<code>tipc_ref_valid()</code>	– Validate a reference to a TIPC port	201
<code>tipc_send()</code>	– Send data over TIPC connection	202
<code>tipc_send2name()</code>	– Send data to a named TIPC port	203
<code>tipc_send2port()</code>	– Send data to a TIPC port ID	204
<code>tipc_send_buf()</code>	– Send message buffer over TIPC connection	205
<code>tipc_send_buf2name()</code>	– Send message buffer to a named TIPC port	206
<code>tipc_send_buf2port()</code>	– Send message buffer to a TIPC port ID	207
<code>tipc_set_portimportance()</code>	– Set importance of TIPC port messages	208
<code>tipc_set_portunreliable()</code>	– Set reliability of TIPC port messages	208

<code>tipc_set_portunreturnable()</code>	– Set returnability of TIPC port messages	209
<code>tipc_shutdown()</code>	– Disconnect a TIPC port from its peer	210
<code>tipc_withdraw()</code>	– Remove a name or name sequence from a TIPC port	210

tipc_attach()

NAME `tipc_attach()` – Register a TIPC user (native API only)

SYNOPSIS

```
int tipc_attach
(
    unsigned int *   userref,      /* returned TIPC user id */
    tipc_mode_event cb,          /* callback routine */
    void *           usr_handle    /* argument to callback routine */
)
```

DESCRIPTION This routine adds a user of TIPC to the list of users. A callback can be specified that will be called whenever the operating mode of TIPC changes from **TIPC_NOT_RUNNING** to **TIPC_NODE_MODE** or **TIPC_NET_MODE**, or vice versa; the callback is also called immediately if TIPC is running in **TIPC_NODE_MODE** or **TIPC_NET_MODE** at the time **tipc_attach()** is called.

NOTE: This routine may be called when TIPC is inactive.

Parameters:

userref

TIPC userid assigned to the newly registered user. This value must be used when deregistering a user via **tipc_detach()**.

cb

tipc_mode_event - TIPC operating mode change callback

```
typedef void (*tipc_mode_event)
(
    void *           *usr_handle, /* user defined handle */
    int              mode,        /* new operating mode */
    u32              addr         /* address of this node */
)
```

This is a user-supplied callback routine that will be called in the event that the operating mode of TIPC changes. The existing operating modes are **TIPC_NOT_RUNNING**, **TIPC_NODE_MODE**, or **TIPC_NET_MODE**. This routine is registered with the **tipc_attach()** call when registering a TIPC user. There is nothing returned with this callback and any errno can be set within the callback as appropriate to the application.

Parameters:

usr_handle

The user-supplied value that was used in the **tipc_attach** call that registered this callback routine.

mode

The new operating mode of TIPC which will be one of **TIPC_NOT_RUNNING**, **TIPC_NODE_MODE**, or **TIPC_NET_MODE**.

addr
This is the TIPC address of this node (most relevant when the operating mode changes to `TIPC_NET_MODE`).

usr_handle
Value that is passed to the callback function when it is invoked.

RETURNS	<code>TIPC_OK</code> on success, <code>ENOPROTOOPT</code> if TIPC is not running and no callback (cb) is provided, or <code>EBUSY</code> if no more users can be added.
ERRNO	N/A
SEE ALSO	<code>tipc_native</code> , <code>tipc_detach()</code>

tipc_connect2port()

NAME	<code>tipc_connect2port()</code> – Associate a TIPC port with its peer (native API only)
SYNOPSIS	<pre>int tipc_connect2port (u32 portref, /* port reference */ struct tipc_portid const *port /* port ID of peer port */)</pre>
DESCRIPTION	This routine associates a TIPC port with the peer port to which it is connected.
CAUTION	This routine is provided for advanced TIPC users, and can not be used to initiate a typical connect operation, as it does not notify the peer port of the connection attempt; use <code>tipc_send2name()</code> or <code>tipc_send2port()</code> instead. Parameters: <i>portref</i> The reference value of the port. <i>port</i> The port information for the destination port.
RETURNS	<code>TIPC_OK</code> or <code>-EINVAL</code> for an invalid port reference.
ERRNO	N/A
SEE ALSO	<code>tipc_native</code> , <code>tipc_disconnect()</code> , <code>tipc_send2name()</code> , <code>tipc_send2port()</code>

tipc_createport()

NAME `tipc_createport()` – Create a TIPC port (native API only)

SYNOPSIS

```
int tipc_createport
(
    unsigned int      tipc_user, /* TIPC user number */
    void              *user_handle, /* user defined handle */
    unsigned int      importance, /* importance of the port */
    tipc_msg_err_event error_cb, /* cb for any error */
    tipc_named_msg_err_event named_error_cb, /* cb for named msg error */
    tipc_conn_shutdown_event conn_error_cb, /* cb for conn msg error */
    tipc_msg_event     message_cb, /* cb for incoming msg */
    tipc_named_msg_event named_message_cb, /* cb for incoming named msg */
    tipc_conn_msg_event conn_message_cb, /* cb for incoming conn msg */
    tipc_continue_event continue_event_cb, /* cb for congestion abatement */
    /*
        u32              *portref /* port reference returned */
    )
```

DESCRIPTION This routine creates a TIPC port that can send and receive messages using the native API. Any necessary callback routines need to be registered with the port. Not all callback routines are required for any given type of port. All ports require the `tipc_user`, `user_handle`, and `portref` fields to be defined. Connectionless ports typically would also supply a `message_cb` at a minimum. Connection oriented ports typically would also supply a `conn_message_cb`. Note that there is nothing returned by the callback routines and any `errno` that is set by the callback routine is independent of the native API. TIPC will automatically discard the message once the callback routine returns. Any message for which no callback exists will be rejected by TIPC unless it is an errored message which is simply discarded.

Parameters:

tipc_user

TIPC userid that port is associated with. A value of 0 creates an "anonymous" port that is not associated with any registered TIPC user.

user_handle

A user-supplied value that is passed as an argument to the port's callback routines. The callback routines may wish to use this value to identify the port in some manner; for example, it may contain a pointer to a data structure associated with the port.

importance

The importance level of messages sent by the port (one of: `TIPC_LOW_IMPORTANCE`, `TIPC_MEDIUM_IMPORTANCE`, `TIPC_HIGH_IMPORTANCE`, or `TIPC_CRITICAL_IMPORTANCE`). This value can also be changed at any time once the port has been created.

error_cb

tipc_msg_err_event - error handling callback for any **TIPC_DIRECT_MSG** message that has an error code attached to it.

```
typedef void (*tipc_msg_err_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref,     /* destination port */
    struct sk_buff      **buf,       /* pointer to incoming msg */
    unsigned char const *data,       /* pointer to data in incoming msg */
    unsigned int         size,        /* size of data in msg, in bytes */
    int                 reason,       /* error code of incoming msg */
    struct tipc_portid const *attmpt_destid /* originating port */
)
```

This is a user-supplied routine that will be called in the event that a direct (**TIPC_DIRECT_MSG**) message is received for an unconnected port and there is an error code associated with the message.

Parameters:

user_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

buf

A pointer to the incoming message buffer.

data

A pointer to the data in the incoming message buffer.

size

The size of the data, in bytes.

reason

Any error code present in the incoming message.

attmpt_destid

The originating port of the incoming message.

named_error_cb

tipc_named_msg_err_event - error handling callback for any **TIPC_NAMED_MSG** or **TIPC_MCAST_MSG** message with an error code.

```
typedef void (*tipc_named_msg_err_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref,     /* destination port */
    struct sk_buff      **buf,       /* pointer to incoming msg */
    unsigned char const *data,       /* pointer to data in incoming msg */
    unsigned int        size,        /* size of data in msg, in bytes */
    int                 reason,      /* error code of incoming msg */
    struct tipc_portid const *attmpt_dest /* name or name sequence */
)
```

This is a user-supplied routine that will be called in the event that a named (**TIPC_NAMED_MSG**) or multicast (**TIPC_MCAST_MSG**) message is received for an unconnected port containing an error code.

Parameters:

user_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

buf

A pointer to the incoming message buffer.

data

A pointer to the data in the incoming message buffer.

size

The size of the data, in bytes.

reason

Any error code present in the incoming message.

attmpt_dest

The port name or port name sequence used to send this message.

conn_error_cb

tipc_conn_shutdown_event - **tipc_conn_shutdown_event** - connection shutdown callback routine for any incoming connection oriented (**TIPC_CONN_MSG**) message with an error code.

```
typedef void (*tipc_conn_shutdown_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref,     /* destination port */
    struct sk_buff      **buf,       /* pointer to incoming msg */
    unsigned char const *data,       /* pointer to data in incoming msg */
    unsigned int        size,        /* size of data in msg, in bytes */
    int                 reason       /* error code of incoming msg */
)
```

This is a user-supplied routine that will be called in the event that a connection is shut down (a **TIPC_CONN_MSG** message with an error code will be received).

Parameters:

user_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

buf

A pointer to the incoming message buffer.

data

A pointer to the data in the incoming message buffer.

size

The size of the data, in bytes.

reason

Any error code present in the incoming message.

message_cb

tipc_msg_event - an incoming **TIPC_DIRECT_MSG** message is received

```
typedef void (*tipc_msg_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref,      /* destination port */
    struct sk_buff      **buf,        /* pointer to incoming msg */
    unsigned char const *data,        /* pointer to data in incoming msg */
    unsigned int         size,        /* size of data in msg, in bytes */
    int                 importance,   /* incoming message importance */
    struct tipc_portid const *origin /* originating port */
)
```

This is a user-supplied routine that will be called in the event that a direct (**TIPC_DIRECT_MSG**) message is received. This would be the user-supplied receive routine for incoming messages.

Parameters:

user_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

buf

A pointer to the incoming message buffer.

data

A pointer to the data in the incoming message buffer.

size

The size of the data, in bytes.

importance

The message importance, one of **TIPC_LOW_IMPORTANCE**, **TIPC_MEDIUM_IMPORTANCE**, **TIPC_HIGH_IMPORTANCE**, or **TIPC_CRITICAL_IMPORTANCE**.

origin

The originating port of the incoming message.

named_message_cb

tipc_named_msg_event - an incoming **TIPC_NAMED_MSG** or **TIPC_MCAST_MSG** message is received.

```
typedef void (*tipc_named_msg_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref,      /* destination port */
    struct sk_buff      **buf,        /* pointer to incoming msg */
    unsigned char const *data,        /* pointer to data in incoming msg */
    unsigned int         size,         /* size of data in msg, in bytes */
    int                  importance,   /* incoming message importance */
    struct tipc_portid const *origin, /* originating port */
    struct tipc_name_seq const *dest /* destination port */
)
```

This is a user-supplied routine that will be called in the event that a named (**TIPC_NAMED_MSG**) or multicast (**TIPC_MCAST_MSG**) message is received. This would be the user-supplied receive routine for named incoming messages.

Parameters:

usr_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

buf

A pointer to the incoming message buffer.

data

A pointer to the data in the incoming message buffer.

size

The size of the data, in bytes.

importance

The message importance, one of **TIPC_LOW_IMPORTANCE**, **TIPC_MEDIUM_IMPORTANCE**, **TIPC_HIGH_IMPORTANCE**, or **TIPC_CRITICAL_IMPORTANCE**.

origin

The originating port of the incoming message.

dest

This is the destination port for which the incoming message was targeted.

conn_message_cb

tipc_conn_msg_event - a incoming **TIPC_CONN_MSG** message is received.

```
typedef void (*tipc_conn_msg_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref,     /* destination port */
    struct sk_buff      **buf,       /* pointer to incoming msg */
    unsigned char const *data,       /* pointer to data in incoming msg */
    unsigned int         size        /* size of data in msg, in bytes */
)
```

This is a user-supplied routine that will be called in the event that a connection (**TIPC_CONN_MSG**) message is received. This would be the user-supplied receive routine for a connected port.

Parameters:

user_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

buf

A pointer to the incoming message buffer.

data

A pointer to the data in the incoming message buffer.

size

The size of the data, in bytes.

continue_event_cb

tipc_continue_event - this callback is called once port congestion has abated.

```
typedef void (*tipc_continue_event)
(
    void                *usr_handle, /* user defined handle */
    u32                 portref      /* destination port */
)
```

This is a user-supplied routine that will be called once any congestion has abated and the port is once again ready to be used.

Parameters:

user_handle

The user-supplied value that was used when the port was created.

portref

This is the destination port for which the incoming message was targeted.

portref

Pointer to an area that is filled in with the port reference for the newly created port.

RETURNS **TIPC_OK** or **-ENOMEM** if a port could not be created.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_deleteport()**

tipc_deleteport()

NAME **tipc_deleteport()** – Delete a TIPC port (native API only)

SYNOPSIS

```
int tipc_deleteport
(
    u32                portref    /* port reference */
)
```

DESCRIPTION This routine deletes a previously created port. The port can no longer be used to send or receive messages, and all names and name sequences associated with the port are automatically withdrawn.

Parameters:

portref

The port reference for the port to be deleted.

RETURNS **TIPC_OK**, or **-EINVAL** if the port does not exist.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_createport()**

tipc_detach()

NAME	tipc_detach() – Unregister a TIPC user (native API only)
SYNOPSIS	<pre>void tipc_detach (unsigned int userref /* returned TIPC user id */)</pre>
DESCRIPTION	<p>This routine removes a registered TIPC user and deletes all ports created by that user.</p> <p>Parameters:</p> <p><i>userref</i> TIPC userid for user (as assigned by tipc_attach()).</p>
RETURNS	N/A
ERRNO	N/A
SEE ALSO	tipc_native , tipc_attach()

tipc_disconnect()

NAME	tipc_disconnect() – Disassociate a TIPC port with its peer (native API only)
SYNOPSIS	<pre>int tipc_disconnect (u32 portref /* port reference */)</pre>
DESCRIPTION	<p>This routine breaks the association between a TIPC port and the peer port to which it is currently connected.</p>
CAUTION	<p>This routine is provided for advanced TIPC users, and can not be used to initiate a typical disconnect operation, as it does not notify the peer port that the connection has been broken; use tipc_shutdown() instead.</p> <p>Parameters:</p> <p><i>portref</i> The reference value of the port.</p>
RETURNS	TIPC_OK , -EINVAL if the port does not exist, or -ENOTCONN if the port is not connected.

ERRNO N/A

SEE ALSO `tipc_native`, `tipc_connect2port()`, `tipc_shutdown()`

tipc_forward2name()

NAME `tipc_forward2name()` – Forward a message to the named port (native API only - may be obsoleted)

SYNOPSIS

```
int tipc_forward2name
(
    u32                portref,      /* port reference */
    struct tipc_name const *name,     /* name of the dest port */
    u32                domain,       /* domain of name to send to */
    unsigned int        section_count, /* number of message sections */
    struct iovec const  *msg_sect,    /* iovec for the data */
    struct tipc_portid const *origin, /* port information storage */
    unsigned int        importance    /* importance of message */
)
```

DESCRIPTION This routine takes the iovec describing the outgoing message and sends it to the named port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested.

Use this routine to send a message described by an iovec to a port by name and specify the originating port id.

Note that this routine may be obsoleted in future releases of TIPC.

Parameters:

portref

The port reference value.

name

The port name to send the message buffer.

domain

The domain of the name to send to. This must be one of **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**.

section_count

The number of message sections to be sent.

msg_sect

The iovec pointing to the message segments.

origin

Originating port information.

importance

The new importance value. Must be one of **TIPC_LOW_IMPORTANCE**, **TIPC_MEDIUM_IMPORTANCE**, **TIPC_HIGH_IMPORTANCE**, or **TIPC_CRITICAL_IMPORTANCE**.

RETURNS number of bytes sent, **-EINVAL** for an invalid port reference, or **-ELINKCONG** if the port is congested.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_forward2port()**, **tipc_forward_buf2name()**, **tipc_forward_buf2port()**

tipc_forward2port()

NAME **tipc_forward2port()** – Forward a message to a port (native API only - may be obsoleted)

SYNOPSIS

```
int tipc_forward2port
(
    u32                portref,      /* port reference */
    struct tipc_portid const *dest,  /* dest port information */
    unsigned int       num_sect,    /* number of message sections */
    struct iovec const *msg_sect,   /* iovector for the data */
    struct tipc_portid const *orig,  /* port information storage */
    unsigned int       importance    /* importance of message */
)
```

DESCRIPTION This routine takes the iovector describing the outgoing message and sends it to the specified port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested. The originating port and port importance are defaulted to that of the portref port.

Use this routine to send a direct message described by an iovector to a port by id and also specify the originating port.

Note that this routine may be obsoleted in future releases of TIPC.

Parameters:

portref

The port reference value.

dest

The port id to send the message buffer.

num_sect

The number of message sections to be sent.

msg_sect

The iovector pointing to the message segments.

origin

Originating port information.

importance

The new importance value. Must be one of **TIPC_LOW_IMPORTANCE**, **TIPC_MEDIUM_IMPORTANCE**, **TIPC_HIGH_IMPORTANCE**, or **TIPC_CRITICAL_IMPORTANCE**. **TIPC_PORT_IMPORTANCE** can also be used which defaults to the current importance setting of the port.

RETURNS	number of bytes sent, -EINVAL for an invalid port reference, or -ELINKCONG if the port is congested.
ERRNO	N/A
SEE ALSO	tipc_native , tipc_forward2name() , tipc_forward_buf2name() , tipc_forward_buf2port()

C

tipc_forward_buf2name()

NAME **tipc_forward_buf2name()** – Forward a buffer to the named port (native API only - may be obsoleted)

SYNOPSIS

```
int tipc_forward_buf2name
(
    u32                portref,      /* port reference */
    struct tipc_name const *name,    /* name of the dest port */
    u32                domain,      /* domain of name to send to */
    struct sk_buff      *buf,        /* buffer to send to peer */
    unsigned int        dsz,         /* size of the buffer */
    struct tipc_portid const *orig,  /* port information storage */
    unsigned int        importance  /* importance of message */
)
```

DESCRIPTION This routine takes the iovector describing the outgoing message and sends it to the named port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested.

Use this routine to send a buffer to a port by name and specify the originating port id.

Note that this routine may be obsoleted in future releases of TIPC.

Parameters:

portref

The port reference value.

name

The port name to send the message buffer.

domain

The domain of the name to send to. This must be one of **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**.

buf

The message buffer to send.

dsz

The size of the message to send.

orig

Originating port information.

importance

The new importance value. Must be one of **TIPC_LOW_IMPORTANCE**, **TIPC_MEDIUM_IMPORTANCE**, **TIPC_HIGH_IMPORTANCE**, or **TIPC_CRITICAL_IMPORTANCE**.

RETURNS	number of bytes sent, -EINVAL for an invalid port reference, or -ELINKCONG if the port is congested.
ERRNO	N/A
SEE ALSO	tipc_native , tipc_forward2name() , tipc_forward2port() , tipc_forward_buf2port()

tipc_forward_buf2port()

NAME **tipc_forward_buf2port()** – Forward a buffer to a port (native API only - may be obsoleted)

SYNOPSIS

```
int tipc_send_buf2port
(
    u32                portref,          /* port reference */
    struct tipc_portid const *dest,      /* dest port information */
    struct sk_buff      *buf,            /* buffer to send to peer */
    unsigned int        dsz,             /* size of the buffer */
    struct tipc_portid const *orig,      /* port information storage */
    unsigned int        importance       /* importance of message */
)
```


DESCRIPTION	<p>This routine takes the iovector describing the outgoing message and sends it to the specified port if possible. If the message cannot be sent, then either -ELINKCONG is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested. The originating port and port importance are defaulted to that of the portref port.</p> <p>This routine is the equivalent of <code>tipc_forward_buf2port</code> using the port reference and our own node id as the originating port; and the existing importance of the port.</p> <p>Use this routine to send a buffer to a port by id and also specify the originating port.</p> <p>Note that this routine may be obsoleted in future releases of TIPC.</p> <p>Parameters:</p> <p><i>portref</i> The port reference value.</p> <p><i>dest</i> The port id to send the message buffer.</p> <p><i>buf</i> The message buffer to send.</p> <p><i>dsz</i> The size of the message to send.</p> <p><i>origin</i> Originating port information.</p> <p><i>importance</i> The new importance value. Must be one of TIPC_LOW_IMPORTANCE, TIPC_MEDIUM_IMPORTANCE, TIPC_HIGH_IMPORTANCE, or TIPC_CRITICAL_IMPORTANCE. TIPC_PORT_IMPORTANCE can also be used which defaults to the current importance setting of the port.</p>
RETURNS	number of bytes sent, -EINVAL for an invalid port reference, or -ELINKCONG if the port is congested.
ERRNO	N/A
SEE ALSO	<code>tipc_native</code> , <code>tipc_forward2name()</code> , <code>tipc_forward2port()</code> , <code>tipc_forward_buf2name()</code>

tipc_get_addr()

NAME	tipc_get_addr() – Get the network address for this node (native API only)
SYNOPSIS	<code>u32 tipc_get_addr (void)</code>
DESCRIPTION	This routine returns the tipc network address (i.e. Z.C.N value) of this node. This value can be converted to its constituent parts using the tipc_node() , tipc_cluster() , and tipc_zone() routines.
RETURNS	A 32-bit value representing the Z.C.N address of this node.
ERRNO	N/A
SEE ALSO	tipc_native

tipc_get_mode()

NAME	tipc_get_mode() – Get operating mode of TIPC (native API only)
SYNOPSIS	<code>int tipc_get_mode (void)</code>
DESCRIPTION	This routine returns the current operating mode of TIPC, which can be one of: TIPC_NOT_RUNNING - TIPC is not active TIPC_NODE_MODE - TIPC is active, but limited to intra-node messaging TIPC_NET_MODE - TIPC is active and capable of inter-node messaging
RETURNS	TIPC_NOT_RUNNING , TIPC_NODE_MODE , or TIPC_NET_MODE .
ERRNO	N/A
SEE ALSO	tipc_native

tipc_isconnected()

NAME	tipc_isconnected() – Determine if a TIPC port is connected (native API only)
SYNOPSIS	<pre>int tipc_isconnected (u32 portref, /* port reference */ unsigned int *isconnected /* returned connection status */)</pre>
DESCRIPTION	<p>This routine determines if the specified TIPC port is currently connected to another port.</p> <p>Parameters:</p> <p><i>portref</i> The reference value of the port.</p> <p><i>isconnected</i> Pointer to an area to store the connection status of the port (1 = connected, 0 = not connected).</p>
RETURNS	TIPC_OK or -EINVAL if the port does not exist.
ERRNO	N/A
SEE ALSO	tipc_native

tipc_ispublished()

NAME	tipc_ispublished() – Determine if a TIPC name exists (native API only)
SYNOPSIS	<pre>int tipc_ispublished (struct tipc_name const *name /* port name to check */)</pre>
DESCRIPTION	<p>This routine determines if the specified port name (i.e. {type, instance} value) is known to TIPC. If the port name is known, an application will be able to send messages to the port(s) having that name.</p> <p>Parameters:</p> <p><i>name</i> Pointer to the port name of interest.</p>

RETURNS	1 if the name is published, 0 otherwise.
ERRNO	N/A
SEE ALSO	tipc_native

tipc_multicast()

NAME **tipc_multicast()** – Multicast data to a set of named TIPC ports (native API only)

SYNOPSIS

```
int tipc_multicast
(
    u32                portref,      /* port reference */
    struct tipc_name_seq const *name_seq /* name sequence for the port */
    u32                domain,      /* domain of name to send to */
    unsigned int        section_count, /* number of message sections */
    struct iovec const  *msg         /* iovec for the data */
)
```

DESCRIPTION This routine takes the iovec describing the outgoing message and sends it to any port with a matching name or name sequence within the domain specified. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested. The originating port and port importance are defaulted to that of the portref port.

Use this routine to send a message described by an iovec to a port name sequence used for multicasting.

set up connection.

Parameters:

portref
The port reference value.

name_seq
The name sequence to multicast a message to.

domain
The domain of the name to send to. This must be one of **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**.

section_count
The number of message sections to be sent.

msg
The iovec pointing to the message segments.

RETURNS	number of bytes sent, -EINVAL for an invalid port reference, -ELINKCONG if the port is congested, or -ENOMEM if the buffer cannot be cloned.
ERRNO	N/A
SEE ALSO	tipc_native

tipc_ownidentity()

NAME **tipc_ownidentity()** – Get port ID of TIPC port (native API only)

SYNOPSIS

```
int tipc_ownidentity
(
    u32          portref,      /* port reference */
    struct tipc_portid *port    /* returned port ID */
)
```

DESCRIPTION This routine obtains the port ID of the specified TIPC port.

NOTE This routine does not validate that the specified port actually exists.

Parameters:

portref
 The reference value of the port.

port
 Pointer to an area that is filled in with the port ID of the specified port.

RETURNS **TIPC_OK**

ERRNO N/A

SEE ALSO **tipc_native**

tipc_peer()

NAME	tipc_peer() – Get the port ID of a TIPC port's peer (native API only)
SYNOPSIS	<pre>int tipc_peer (u32 ref, /* port reference */ struct tipc_portid *port /* returned port ID */)</pre>
DESCRIPTION	<p>This routine returns the port ID of the peer port to which the specified TIPC port is currently connected.</p> <p>Parameters:</p> <p><i>ref</i> The reference value of the port.</p> <p><i>port</i> Pointer to area for the port ID of the port's peer.</p>
RETURNS	TIPC_OK , -EINVAL for an invalid port reference, or -ENOTCONN for an unconnected port.
ERRNO	N/A
SEE ALSO	tipc_native

tipc_portimportance()

NAME	tipc_portimportance() – Get importance of TIPC port messages (native API only)
SYNOPSIS	<pre>int tipc_portimportance (u32 portref, /* port reference */ unsigned int *importance /* returned port importance */)</pre>
DESCRIPTION	<p>This routine obtains the importance level of messages sent by a TIPC port.</p> <p>Parameters:</p> <p><i>portref</i> The reference value of the port.</p> <p><i>importance</i> Pointer to an area that is filled in with the importance value.</p>

RETURNS TIPC_OK, or -EINVAL if the port does not exist.

ERRNO N/A

SEE ALSO *tipc_native*, *tipc_set_portimportance()*

tipc_portunreliable()

NAME *tipc_portunreliable()* – Get reliability of TIPC port messages (native API only)

SYNOPSIS

```
int tipc_portunreliable
(
    u32          portref,      /* port reference */
    unsigned int *isunreliable /* returned reliability setting */
)
```

DESCRIPTION This routine indicates if messages sent by the port are being sent in an unreliable manner (i.e. the messages are silently discarded if congestion occurs).

Parameters:

portref

The reference value of the port.

isunreliable

Pointer to the area where the reliability setting for the port is stored (0 = send reliably, 1 = send unreliably).

RETURNS TIPC_OK, or -EINVAL if the port does not exist.

ERRNO N/A

SEE ALSO *tipc_native*, *tipc_set_portunreliable()*

tipc_portunreturnable()

NAME	tipc_portunreturnable() – Get returnability of TIPC port messages (native API only)
SYNOPSIS	<pre>int tipc_portunreturnable (u32 portref, /* port reference */ unsigned int *isunreturnable /* returned returnability setting */)</pre>
DESCRIPTION	<p>This routine indicates if messages sent by the port are being sent in a non-returnable manner (i.e. the messages are silently discarded if they cannot be delivered to the specified destination).</p> <p>Parameters:</p> <p><i>portref</i> The reference value of the port.</p> <p><i>isunreturnable</i> Pointer to the area where the returnability setting for the port is stored (0 = messages are returnable, 1 = messages are non-returnable).</p>
RETURNS	TIPC_OK , or -EINVAL if the port does not exist.
ERRNO	N/A
SEE ALSO	tipc_native , tipc_set_portunreturnable()

tipc_publish()

NAME	tipc_publish() – Add a name or name sequence to a TIPC port (native API only)
SYNOPSIS	<pre>int tipc_publish (u32 portref, /* port reference */ unsigned int scope, /* scope of the publication */ struct tipc_name_seq const *name_seq /* name sequence for the port */)</pre>
DESCRIPTION	<p>This routine adds a TIPC name sequence (i.e. {type, lower bound, upper bound} value) to the set of names associated with a TIPC port. It can also be used to add a TIPC name (i.e. {type, instance} value) by specifying a name sequence in which the lower bound and upper bound are the same.</p>

The name sequence will be publicized to all nodes in the TIPC network that lie within the specified publication scope.

Parameters:

portref
 The reference value of the port.

scope
 The publication scope for the name sequence (one of: **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**).

name_seq
 The name sequence to be associate with the TIPC port.

RETURNS **TIPC_OK**, or **-EINVAL** if the port does not exist, the port is connected, the name sequence is invalid, the type is reserved, or the scope is invalid, or **-EADDRINUSE** if the reference values have wrapped.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_withdraw()**

C

tipc_ref_valid()

NAME **tipc_ref_valid()** – Validate a reference to a TIPC port (native API only)

SYNOPSIS

```
int tipc_ref_valid
(
    u32                                portref    /* port reference */
)
```

DESCRIPTION This routine determines if the TIPC port associated with the specified port reference currently exists.

Parameters:

portref
 The reference value of the port.

RETURNS 1 if the port exists, 0 otherwise

ERRNO N/A

SEE ALSO **tipc_native**

tipc_send()

NAME `tipc_send()` – Send data over TIPC connection (native API only)

SYNOPSIS

```
int tipc_send
(
    u32                portref,    /* port reference */
    unsigned int       num_sect,   /* number of message sections */
    struct iovec const *msg_sect   /* iovector for the data */
)
```

DESCRIPTION This routine takes the iovector describing the outgoing message and sends it to the connected peer port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested.

Use this routine to send a message described by an iovector to a previously set up connection.

Parameters:

portref
The port reference value.

num_sect
The number of message sections to be sent.

msg_sect
The iovector pointing to the message segments.

RETURNS number of bytes sent, **-EINVAL** for an invalid port reference, or **-ELINKCONG** if the port is congested.

ERRNO N/A

SEE ALSO `tipc_native`, `tipc_send2name()`, `tipc_send2port()`, `tipc_send_buf()`, `tipc_send_buf2name()`, `tipc_send_buf2port()`

tipc_send2name()

NAME `tipc_send2name()` – Send data to a named TIPC port (native API only)

SYNOPSIS

```
int tipc_send2name
(
    u32                portref,      /* port reference */
    struct tipc_name const *name,    /* name of the dest port */
    u32                domain,      /* domain of name to send to */
    unsigned int        num_sect,   /* number of message sections */
    struct iovec const  *msg_sect   /* iovector for the data */
)
```

DESCRIPTION This routine takes the iovector describing the outgoing message and sends it to the named port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested.

This routine is the equivalent of `tipc_forward2name` using the port reference and our own node id as the originating port.

Use this routine to send a message described by an iovector to a port by name.

Parameters:

portref

The port reference value.

name

The port name to send the message buffer.

domain

The domain of the name to send to. This must be one of **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**.

num_sect

The number of message sections to be sent.

msg_sect

The iovector pointing to the message segments.

RETURNS number of bytes sent, **-EINVAL** for an invalid port reference, or **-ELINKCONG** if the port is congested.

ERRNO N/A

SEE ALSO `tipc_native`, `tipc_send()`, `tipc_send2port()`, `tipc_send_buf()`, `tipc_send_buf2name()`, `tipc_send_buf2port()`

tipc_send2port()

NAME	tipc_send2port() – Send data to a TIPC port ID (native API only)
SYNOPSIS	<pre>int tipc_send2port (u32 portref, /* port reference */ struct tipc_portid const *dest, /* dest port information */ unsigned int num_sect, /* number of message sections */ struct iovec const *msg_sect /* iovector for the data */)</pre>
DESCRIPTION	<p>This routine takes the iovector describing the outgoing message and sends it to the specified port if possible. If the message cannot be sent, then either -ELINKCONG is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested. The originating port and port importance are defaulted to that of the portref port.</p> <p>This routine is the equivalent of <code>tipc_forward2port</code> using the port reference and our own node id as the originating port; and the existing importance of the port.</p> <p>Use this routine to send a direct message described by an iovector to a port by id from the port referenced.</p> <p>Parameters:</p> <p><i>portref</i> The port reference value.</p> <p><i>dest</i> The port id to send the message buffer.</p> <p><i>num_sect</i> The number of message sections to be sent.</p> <p><i>msg_sect</i> The iovector pointing to the message segments.</p>
RETURNS	number of bytes sent, -EINVAL for an invalid port reference, or -ELINKCONG if the port is congested.
ERRNO	N/A
SEE ALSO	tipc_native , tipc_send() , tipc_send2name() , tipc_send_buf() , tipc_send_buf2name() , tipc_send_buf2port()

tipc_send_buf()

NAME `tipc_send_buf()` – Send message buffer over TIPC connection (native API only)

SYNOPSIS

```
int tipc_send_buf
(
    u32                portref,    /* port reference */
    struct sk_buff      *buf,      /* buffer to send to peer */
    unsigned int        dsz        /* size of the buffer */
)
```

DESCRIPTION This routine takes the buffer and size specified in the parameters and sends it to the connected peer port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested.

Use this routine to send a message buffer to a previously set up connection.

Parameters:

portref
 The port reference value.

buf
 The message buffer to send.

dsz
 The size of the message to send.

RETURNS number of bytes sent, **-EINVAL** for an invalid port reference, **-ELINKCONG** if the port is congested, or **-ENOMEM** if the buffer cannot be cloned.

ERRNO N/A

SEE ALSO `tipc_native`, `tipc_send()`, `tipc_send2name()`, `tipc_send2port()`, `tipc_send_buf2name()`, `tipc_send_buf2port()`

tipc_send_buf2name()

NAME `tipc_send_buf2name()` – Send message buffer to a named TIPC port (native API only)

SYNOPSIS

```
int tipc_send_buf2name
(
    u32                portref,    /* port reference */
    struct tipc_name const *name,  /* name of the dest port */
    u32                domain,    /* domain of name to send to */
    struct sk_buff      *buf,     /* buffer to send to peer */
    unsigned int        dsz       /* size of the buffer */
)
```

DESCRIPTION This routine takes the buffer and size specified in the parameters and sends it to the connected peer port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested.

Use this routine to send a buffer to a port by name.

Parameters:

portref
The port reference value.

name
The port name to send the message buffer.

domain
The domain of the name to send to. This must be one of **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**.

buf
The message buffer to send.

dsz
The size of the message to send.

RETURNS number of bytes sent, **-EINVAL** for an invalid port reference, **-ELINKCONG** if the port is congested, or **-ENOMEM** if the buffer cannot be cloned.

ERRNO N/A

SEE ALSO `tipc_native`, `tipc_send()`, `tipc_send2name()`, `tipc_send2port()`, `tipc_send_buf()`, `tipc_send_buf2port()`

tipc_send_buf2port()

NAME `tipc_send_buf2port()` – Send message buffer to a TIPC port ID (native API only)

SYNOPSIS

```
int tipc_send_buf2port
(
    u32                portref,      /* port reference */
    struct tipc_portid const *dest,   /* dest port information */
    struct sk_buff      *buf,        /* buffer to send to peer */
    unsigned int        dsz          /* size of the buffer */
)
```

DESCRIPTION This routine takes the iovec describing the outgoing message and sends it to the specified port if possible. If the message cannot be sent, then either **-ELINKCONG** is returned if the port is reliable, or the message size is calculated and returned for an unreliable port. In the event of congestion, the port is marked as congested. The originating port and port importance are defaulted to that of the portref port.

This routine is the equivalent of `tipc_forward_buf2port` using the port reference and our own node id as the originating port; and the existing importance of the port.

Use this routine to send a buffer to a port by id from the port referenced.

Parameters:

portref
 The port reference value.

dest
 The port id to send the message buffer.

buf
 The message buffer to send.

dsz
 The size of the message to send.

RETURNS number of bytes sent, **-EINVAL** for an invalid port reference, or **-ELINKCONG** if the port is congested.

ERRNO N/A

SEE ALSO `tipc_native`, `tipc_send()`, `tipc_send2name()`, `tipc_send2port()`, `tipc_send_buf()`, `tipc_send_buf2name()`

tipc_set_portimportance()

NAME	tipc_set_portimportance() – Set importance of TIPC port messages (native API only)
SYNOPSIS	<pre>int tipc_set_portimportance (u32 portref, /* port reference */ unsigned int importance /* new port importance */)</pre>
DESCRIPTION	<p>This routine sets the importance level of messages sent by a TIPC port. Note that this setting will have an influence on the number of messages that may be queued up by the receiver if the receiver is running a socket layer.</p> <p>By default, a TIPC port sends messages using the importance level specified by the user when the port was created.</p> <p>Parameters:</p> <p><i>portref</i> The reference value of the port.</p> <p><i>importance</i> The new importance value (TIPC_LOW_IMPORTANCE, TIPC_MEDIUM_IMPORTANCE, TIPC_HIGH_IMPORTANCE, or TIPC_CRITICAL_IMPORTANCE).</p>
RETURNS	TIPC_OK, or -EINVAL if the port does not exist or an invalid importance level is specified.
ERRNO	N/A
SEE ALSO	tipc_native, tipc_portimportance()

tipc_set_portunreliable()

NAME	tipc_set_portunreliable() – Set reliability of TIPC port messages (native API only)
SYNOPSIS	<pre>int tipc_set_portunreliable (u32 portref, /* port reference */ unsigned int isunreliable /* new reliability setting */)</pre>
DESCRIPTION	<p>This routine specifies if messages sent by the port are to be sent in an unreliable manner (i.e. the messages will be silently discarded if congestion occurs).</p>

By default, a TIPC port sends messages reliably.

Parameters:

portref

The reference value of the port.

isunreliable

The new reliability setting (0 = send reliably, non-zero = send unreliably).

RETURNS **TIPC_OK**, or **-EINVAL** if the port does not exist.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_portunreliable()**

tipc_set_portunreturnable()

NAME **tipc_set_portunreturnable()** – Set returnability of TIPC port messages (native API only)

SYNOPSIS

```
int tipc_set_portunreturnable
(
    u32          portref,          /* port reference */
    unsigned int  isunreturnable /* new returnability setting */
)
```

DESCRIPTION This routine specifies if messages sent by the port are to be sent in an non-returnable manner (i.e. the messages will be silently discarded if they cannot be delivered to the specified destination).

By default, a TIPC port sends returnable messages.

Parameters:

portref

The reference value of the port.

isunreturnable

The new returnability setting (0 = messages are returnable, non-zero = messages are non-returnable).

RETURNS **TIPC_OK**, or **-EINVAL** if the port does not exist.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_portunreturnable()**

tipc_shutdown()

NAME	tipc_shutdown() – Disconnect a TIPC port from its peer (native API only)
SYNOPSIS	<pre>int tipc_shutdown (u32 ref /* port reference */)</pre>
DESCRIPTION	<p>This routine gracefully terminates a connection between a TIPC port and its peer. The TIPC port is disconnected, and its peer is notified that the connection has been terminated.</p> <p>Parameters:</p> <p><i>ref</i> The reference value of the port.</p>
RETURNS	TIPC_OK , -EINVAL if the port does not exist, or -ENOTCONN if the port is not connected.
ERRNO	N/A
SEE ALSO	tipc_native

tipc_withdraw()

NAME	tipc_withdraw() – Remove a name or name sequence from a TIPC port (native API only)
SYNOPSIS	<pre>int tipc_withdraw (u32 portref, /* port reference */ unsigned int scope, /* scope of the publication */ struct tipc_name_seq const *name_seq /* name sequence for the port */)</pre>
DESCRIPTION	<p>This routine removes a TIPC name sequence (i.e. {type, lower bound, upper bound} value) from the set of names associated with a TIPC port. It can also be used to remove a TIPC name (i.e. {type, instance} value) by specifying a name sequence in which the lower bound and upper bound are the same. The specified name sequence and scope values must match those of an existing publication.</p> <p>This routine can also remove all published names in a single operation by specifying a NULL name sequence.</p> <p>The specified name sequence(s) will also be unpublicized on all nodes in the TIPC network that have been previously notified of the publication.</p>

Parameters:

portref

The reference value of the port.

scope

The publication scope for the name sequence (one of: **TIPC_NODE_SCOPE**, **TIPC_CLUSTER_SCOPE**, or **TIPC_ZONE_SCOPE**). This value is ignored when removing all names from the port.

name_seq

The name sequence to remove from the port's set of names. Specifying **NULL** will remove all published names for this port.

RETURNS **TIPC_OK**, or **-EINVAL** if the port or specified name sequence does not exist.

ERRNO N/A

SEE ALSO **tipc_native**, **tipc_publish()**

D

Header File Definitions

[D.1 Introduction 213](#)

[D.2 Definitions 214](#)

D.1 Introduction

This appendix lists public type definitions, defines, and structures contained in the Wind River TIPC header file:

installDir/target/h/tipc/tipc.h

You need to include this file (**#include <ipc/tipc.h>**) in all TIPC applications.

D.2 Definitions

```
/* Scalar data types used by TIPC (WRS) */

typedef unsigned char    __u8, uchar;
typedef char            __s8;
typedef unsigned short   __u16;
typedef short           __s16;
typedef unsigned int     __u32, uint;
typedef int              __s32;

typedef unsigned char    u8;
typedef char             s8;
typedef unsigned short   u16;
typedef short            s16;
typedef unsigned int     u32;
typedef int              s32;

/*
 * TIPC addressing primitives
 *
 * (Uses macros rather than inline functions to avoid compiler warnings)
 */

struct tipc_portid {
    __u32 ref;
    __u32 node;
};

struct tipc_name {
    __u32 type;
    __u32 instance;
};

struct tipc_name_seq {
    __u32 type;
    __u32 lower;
    __u32 upper;
};

#define tipc_addr(Z,C,N) (((Z)<<24)|((C)<<12)|(N))
#define tipc_zone(A)     ((A)>> 24)
#define tipc_cluster(A)  (((A)>> 12) & 0xfff)
#define tipc_node(A)     ((A) & 0xfff)

/*
 * Task defines for handler.c and others that need to know the PRIORITY
 */

#define TIPC_TASK_PRIORITY (52)
#define TIPC_TASK_OPTIONS (0)
#define TIPC_TASK_STACKSIZE (5000)
```

```

/*
 * Application-accessible port name types
 */

#define TIPC_NET_EVENTS          0      /* network event subscription name type */
#define TIPC_TOP_SRV            1      /* topology service name type */
#define TIPC_RESERVED_TYPES     64     /* lowest user-publishable name type */

/*
 * Publication scopes when binding port names and port name sequences
 */

#define TIPC_ZONE_SCOPE          1
#define TIPC_CLUSTER_SCOPE       2
#define TIPC_NODE_SCOPE          3

/*
 * Limiting values for messages
 */

#define TIPC_MAX_USER_MSG_SIZE 66000

/*
 * Message importance levels
 */

#define TIPC_LOW_IMPORTANCE      0      /* default */
#define TIPC_MEDIUM_IMPORTANCE  1
#define TIPC_HIGH_IMPORTANCE    2
#define TIPC_CRITICAL_IMPORTANCE 3

/*
 * Msg rejection/connection shutdown reasons
 */

#define TIPC_OK                   0
#define TIPC_ERR_NO_NAME         1
#define TIPC_ERR_NO_PORT        2
#define TIPC_ERR_NO_NODE        3
#define TIPC_ERR_OVERLOAD        4
#define TIPC_CONN_SHUTDOWN      5

/*
 * TIPC topology subscription service definitions
 */

#define TIPC_SUB_PORTS           0x01   /* filter for port availability */
#define TIPC_SUB_SERVICE        0x02   /* filter for service availability */
#define 0
/* The following filter options are not currently implemented */
#define TIPC_SUB_NO_BIND_EVTS   0x04   /* filter out "publish" events */
#define TIPC_SUB_NO_UNBIND_EVTS 0x08   /* filter out "withdraw" events */
#define TIPC_SUB_SINGLE_EVT     0x10   /* expire after first event */
#endif

```

```
#define TIPC_WAIT_FOREVER    ((__u32)~0) /* timeout for permanent subscription */

struct tipc_subscr {
    struct tipc_name_seq seq;          /* name sequence of interest */
    __u32 timeout;                    /* subscription duration (in ms) */
    __u32 filter;                     /* bitmask of filter options */
    char usr_handle[8];               /* available for subscriber use */
};

#define TIPC_PUBLISHED        1        /* publication event */
#define TIPC_WITHDRAWN       2        /* withdraw event */
#define TIPC_SUBSCR_TIMEOUT   3        /* subscription timeout event */

struct tipc_event {
    __u32 event;                      /* event type */
    __u32 found_lower;               /* matching name seq instances */
    __u32 found_upper;               /* " " " " " */
    struct tipc_portid port;         /* associated port */
    struct tipc_subscr s;            /* associated subscription */
};

/*
 * Socket API
 * -----
 */

#ifndef AF_TIPC
#define AF_TIPC 33 /* SOCK_STREAM, SOCK_SEQPACKET, SOCK_RDM, SOCK_DGRAM */
#endif

#define TIPC_ADDR_NAMESEQ      1
#define TIPC_ADDR_MCAST       1
#define TIPC_ADDR_NAME        2
#define TIPC_ADDR_ID           3

struct sockaddr_tipc {
    unsigned char  addrlen;           /* 16 */
    unsigned char  family;            /* AF_TIPC */
    unsigned char  addrtype;          /* TIPC_ADDR_XXX */
    unsigned char  scope;             /* used with bind */
    union {
        struct tipc_portid id;        /* if TIPC_ADDR_ID */
        struct tipc_name_seq nameseq; /* if TIPC_ADDR_NAMESEQ/_MCAST */
        struct {
            struct tipc_name name;    /* if TIPC_ADDR_NAME */
            __u32 domain;             /* 0: own zone; used w/ connect, sendto */
        } name;
    } addr;
};
```



```
/*
 * Ancillary data objects supported by recvmmsg()
 */

#define TIPC_ERRINFO      1          /* error info */
#define TIPC_RETDATA      2          /* returned data */
#define TIPC_DESTNAME     3          /* destination name */

/*
 * TIPC-specific socket option values
 */

#define SOL_TIPC           50        /* TIPC socket option level */
#define TIPC_IMPORTANCE    127       /* Default: TIPC_LOW_IMPORTANCE */
#define TIPC_SRC_DROPPABLE 128       /* Default: 0 (resend congested msg) */
#define TIPC_DEST_DROPPABLE 129      /* Default: based on socket type */
#define TIPC_CONN_TIMEOUT  130       /* Default: 8000 (ms) */
```


E

Sample TIPC Application

[E.1 Introduction 219](#)

[E.2 TIPC Inventory Simulation 220](#)

E.1 Introduction

This appendix provides a sample application illustrating the use of the Wind River TIPC API. The application is a demonstration program, **tipcInventorySim_VxWorks.c**, that simulates a store with items for sale and customers who enter the store to purchase items.

The demonstration program is available online:

`installDir/vxworks-6.x/target/src/demo/tipcInventorySim_VxWorks.c`

If you include the **TIPC inventory simulation demo** (INCLUDE_TIPC_IS) build component in your build, the code for the inventory simulation is brought into your VxWorks Image Project as a downloadable kernel project, ready for compilation.

E.2 TIPC Inventory Simulation

The following description of a sample TIPC program is slightly modified from the original text in `tipcInventorySim_VxWorks.c`. The source code is unchanged.

E.2.1 Description

The file `tipcInventorySim_VxWorks.c` contains a demonstration program that illustrates the way TIPC can be used to support distributed applications. It takes advantage of TIPC's reliable connection-oriented messaging, its port-naming (functional-addressing) capability, and its port-name subscription feature.

The program simulates a store that stocks a number of different items. The program randomly creates items for purchase. From time to time a customer enters the store looking for an item. If the item is not available, the customer waits for it for a limited period of time before leaving. The item wanted by each customer and the length of time the customer will wait for it are randomly generated within a fixed range. The interval between the arrival of customers is also random, but there is a limit on the number of customers who can be in the store at one time.

Every item and every customer is implemented as a separate task. An item is available when it creates a socket whose port name identifies the type (see [2.5.3 Functional Addressing](#), p.13) of the item. A customer obtains an item by sending a message to the socket associated with the desired item and receiving a reply, after which both the item and the customer tasks are terminated.

Customers use TIPC's port-name subscription feature (see [2.7 Subscriptions](#), p.18) to determine whether the desired item is available and then use TIPC's *implied connection* capability (see [2.3 Messaging Overview](#), p.9) to establish a connection to the item. If a race condition arises when multiple customers are waiting for the same item, the item is sold to the first customer who connects to it. The remaining customers simply wait for another item of the same type to appear and then try again to purchase it.

The simulation is most effective when it is run on multiple CPUs at the same time. Since the port name used by an item is published throughout the TIPC cluster, customers can obtain the item from another CPU if it is not available locally. The more CPUs involved in the simulation, the more items are available and the less likely a customer is to walk out of the store empty-handed because an item is unavailable.

When the simulation is terminated on a node, it waits for all customers to leave the store and then generates extra customers to purchase any unsold items on the

node. TIPC's name-sequence subscription feature is used to allow the termination code to distinguish between items in its own store and items in other stores.

The following shell commands are available for running the simulation:

- **newSim(*I*)**

Create a new simulation on the current node, where:

$I > 0$ auto-generates items and customers for items **1** to I

$I = 0$ auto-generates items and customers for all possible items

$I < 0$ requires manual generation of items (see **newItem(*I*,*R*)**) and customers (see **newCust(*I*,*R*)**)

- **stopSim**

Halts the simulation on all nodes.

- **startSim**

Resumes the simulation on all nodes.

- **killSim**

Terminates the simulation on this node, only.

- **newItem(*I*,*R*)**

Create one or more items on the current node, where:

$I > 0$ assigns the specified ID to an item

$I = 0$ assigns a randomly generated ID to an item

$R > 0$ creates R items, one at a time

$R = 0$ creates one item

$R < 0$ creates an unlimited number of items, one at a time

- **newCust(*I*,*R*)**

Create one or more customers on the current node, where:

$I > 0$ assigns the specified ID to a customer

$I = 0$ assigns a randomly generated ID to a customer

$R > 0$ creates R customers, one at a time

$R = 0$ creates one customer

$R < 0$ creates an unlimited number of customers, one at a time

While the simulation is halted (**stopSim**), you can use the following commands to get information about the simulation:

- **showSim**

Display status and statistical information about the simulation. Most of the displayed information applies only to the local node.

- **i**

Display all tasks running on this node. Tasks for items have names of the form **item_X**, where *X* is the ID of the item. Tasks for customers have names of the form **custN_X**, where *N* is the ID of the customer and *X* is the ID of the item the customer wants to purchase.

- **tipcConfig "nt"**

If the **tipcConfig** utility is enabled, display port names for all simulated items, including those on other nodes in the TIPC network.

To enable the **tipcConfig** utility, you need to include the **TIPC configuration and display routines (INCLUDE_TIPC_SHOW)** in your project build

E.2.2 Source Code

```
/* tipcInventorySim_VxWorks.c - TIPC distributed inventory sim for VxWorks */

/* Copyright (c) 2004-2005 Wind River Systems, Inc. */

/* includes */

#include <vxWorks.h>
#include <memLib.h>
#include <selectLib.h>
#include <semLib.h>
#include <sockLib.h>
#include <stdio.h>
#include <string.h>
#include <sysLib.h>
#include <taskLib.h>
#include <tickLib.h>
#include <tipc/tipc.h>

/* defines */

#define TIPC_SALES_DEMO_TYPE 75          /* TIPC type # used by items in demo */
#define MSG_SIZE_MAX 50                /* maximum message size (in bytes) */

#define MAX_CUSTOMERS 10                /* maximum # of customers per CPU */
```

```

#define MAX_ITEMS 10                                /* maximum # of items per CPU */

#define DEMO_ITEM_ID_MIN 1                          /* minimum item ID */
#define DEMO_ITEM_ID_MAX 10                         /* maximum item ID */
#define NUM_DEMO_ITEMS (DEMO_ITEM_ID_MAX - DEMO_ITEM_ID_MIN + 1)

#define CUSTOMER_WAIT_MIN 5000                      /* min time a customer will wait (ms) */
#define CUSTOMER_WAIT_MAX 20000                    /* max time a customer will wait (ms) */
#define NEW_CUST_WAIT_MIN 1000                     /* min time before new customer (ms) */
#define NEW_CUST_WAIT_MAX 5000                     /* max time before new customer (ms) */
#define NEW_ITEM_WAIT_MIN 0                        /* min time before new item (ms) */
#define NEW_ITEM_WAIT_MAX 8000                     /* max time before new item (ms) */
#define SIM_STATUS_INTERVAL 10000                  /* time between status displays (ms) */

#define ITEM_GEN_TASK_PRI 110
#define CUSTOMER_GEN_TASK_PRI 110
#define ITEM_TASK_PRI 120
#define CUSTOMER_TASK_PRI 130
#define SIM_STATUS_TASK_PRI 140
#define ITEM_STACK_SIZE 5000
#define CUSTOMER_STACK_SIZE 5000
#define ITEM_GEN_STACK_SIZE 5000
#define CUSTOMER_GEN_STACK_SIZE 5000
#define SIM_STATUS_STACK_SIZE 5000

#define TIPC_BOGUS_SUBSCR_TYPE TIPC_TOP_SRV
#define TIPC_BOGUS_SUBSCR_INST 0

/* locals */

LOCAL int simActive = FALSE;                        /* is simulation created? */
LOCAL int simErrors;                                /* error counter */
LOCAL int simWarnings;                              /* warning counter */

LOCAL SEM_ID semSyncLock;                          /* used to avoid interleaved output
                                                    (and to pause simulation) */
LOCAL SEM_ID semItems;                             /* used to limit # items per CPU */
LOCAL SEM_ID semCustomers;                         /* used to limit # customers per CPU */

LOCAL int tidCustGen;                              /* task ID of customer generator */
LOCAL int tidItemGen;                              /* task ID of item generator */
LOCAL int tidShowSim;                              /* task ID of status display */

LOCAL int ticksPerSec;                             /* helps convert ms to ticks */

LOCAL int itemCount[NUM_DEMO_ITEMS];               /* # of items in stock */
LOCAL int customerCount[NUM_DEMO_ITEMS];           /* # customers waiting for item */
LOCAL int itemsSold[NUM_DEMO_ITEMS];               /* # of items sold */

LOCAL int customerSales;                            /* # customers who left with item */
/* Note: customer sales may not match total items sold
   because customers can buy items from other locations! */
LOCAL int customerExits;                            /* # customers who left w/o item */
LOCAL int customerRetries;                         /* # customers who had to retry */

LOCAL int item0_fd = -1;                           /* socket used to halt simulation */

```

```
LOCAL int doOneLoop = 0;          /* used to clean up the warnings in multi_printf */

/*
 * This macro ensures simultaneous printf's aren't interleaved.
 */

#ifdef _WRS_GNU_VAR_MACROS
#define multi_printf(fmt, arg...) \
    do { \
        semTake (semSyncLock, WAIT_FOREVER); \
        printf (fmt, ## arg); \
        semGive (semSyncLock); \
    } while (doOneLoop)
#else
#define multi_printf(...) \
    do { \
        semTake (semSyncLock, WAIT_FOREVER); \
        printf (__VA_ARGS__); \
        semGive (semSyncLock); \
    } while (doOneLoop)
#endif

/*****
 *
 * randomGet - random number generator
 *
 * This routine returns a random integer in the specified range (inclusive).
 *
 * RETURNS: random value
 *
 */

int randomGet
(
    int minValue,          /* lowest permitted value */
    int maxValue          /* highest permitted value */
)
{
    return (rand() % (maxValue - minValue + 1)) + minValue;
}

/*****
 *
 * showSim - display simulation status
 *
 * This routine prints out the status of the sales demo (on this CPU only).
 *
 * RETURNS: OK or ERROR
 *
 */
```



```

STATUS showSim (void)
{
    int i;                                /* loop counter */

    printf ("\nItem #      :");
    for (i = 0; i < NUM_DEMO_ITEMS; i++)
        printf (" %3d", DEMO_ITEM_ID_MIN + i);
    printf ("\n-----");
    for (i = 0; i < NUM_DEMO_ITEMS; i++)
        printf ("----");
    printf ("\nSold      :");
    for (i = 0; i < NUM_DEMO_ITEMS; i++)
        printf (" %3d", itemsSold[i]);
    printf ("\nIn Stock :");
    for (i = 0; i < NUM_DEMO_ITEMS; i++)
        printf (" %3d", itemCount[i]);
    printf ("\nCustomers:");
    for (i = 0; i < NUM_DEMO_ITEMS; i++)
        printf (" %3d", customerCount[i]);

    printf ("\n\nCustomer totals: sales = %d, walkouts = %d, retries = %d\n\n",
            customerSales, customerExits, customerRetries);
    printf ("Simulation totals: errors = %d, warnings = %d\n\n",
            simErrors, simWarnings);

    return OK;
}

/*****
 *
 * simStatShow - simulation status monitor
 *
 * This routine is the mainline for the simulation status display task.
 * It is also responsible for halting the simulation whenever item 0 exists.
 *
 */

void simStatShow (void)
{
    struct tipc_subscr subscr;             /* subscription info */
    int    sockfd_w;                      /* socket descriptor */
    struct sockaddr_tipc topsrv;           /* topology server socket address */

    memset(&topsrv, 0, sizeof(topsrv));
    topsrv.family    = AF_TIPC;
    topsrv.addrtype  = TIPC_ADDR_NAME;
    topsrv.addr.name.name.type = TIPC_TOP_SRV;
    topsrv.addr.name.name.instance = TIPC_TOP_SRV;

    multi_printf ("Status display task created\n");

    /* Subscribe to watch for item 0 */

    subscr.seq.type  = TIPC_SALES_DEMO_TYPE;
    subscr.seq.lower = 0;
    subscr.seq.upper = 0;

```

```
subscr.timeout    = TIPC_WAIT_FOREVER;
subscr.filter     = TIPC_SUB_SERVICE;
subscr.usr_handle[0] = 0;

/* Create socket to watch for item 0 */

sockfd_w = socket (AF_TIPC, SOCK_SEQPACKET, 0);
if (sockfd_w < 0)
{
    multi_printf ("Can't create socket to watch for item 0\n");
    goto simStatShow_end;
}

if (connect (sockfd_w, (struct sockaddr*)&topsrv, sizeof (topsrv)) < 0)
{
    multi_printf ("show : Can't connect to TOP server\n");
    goto simStatShow_end;
}
if (send (sockfd_w, (char *)&subscr, sizeof (subscr), 0) < 0)
{
    multi_printf ("Can't watch for item 0\n");
    goto simStatShow_end;
}

/* Now loop endlessly */

FOREVER
{
    struct tipc_event event;          /* event from topology server */

    /* Start bogus subscription to force timeout event */

    subscr.seq.type = TIPC_BOGUS_SUBSCR_TYPE;
    subscr.seq.lower = TIPC_BOGUS_SUBSCR_INST;
    subscr.seq.upper = TIPC_BOGUS_SUBSCR_INST;
    subscr.filter    = TIPC_SUB_SERVICE;
    subscr.timeout   = SIM_STATUS_INTERVAL;
    subscr.usr_handle[0]++;

    if (send (sockfd_w, (char *)&subscr, sizeof (subscr), 0) < 0)
    {
        multi_printf ("Can't watch for item 0\n");
        goto simStatShow_end;
    }

    /* Wait until item 0 appears or it's time to print status */

    FOREVER
    {
        if (recv (sockfd_w, (char *)&event, sizeof (event), 0)
            < 0) {
            multi_printf ("Error recv subscription on item 0\n");
            goto simStatShow_end;
        }
    }
}
```

```
        if (event.event == TIPC_PUBLISHED)
            break;
        if ((event.event == TIPC_SUBSCR_TIMEOUT) &&
            (event.s.usr_handle[0] == subscr.usr_handle[0]))
            break;
    }

    /* Quit if simulation has ended */

    if (!simActive)
        break;

    /* Print simulation status */

    semTake (semSyncLock, WAIT_FOREVER);
    showSim ();

    /* Halt simulation as long as item 0 exists */

    if (event.event == TIPC_PUBLISHED)
    {
        printf ("\nSimulation halted\n");
        FOREVER
        {
            if (recv (sockfd_w, (char *)&event, sizeof (event), 0) < 0)
            {
                multi_printf ("Error recv subscription on item 0\n");
                goto simStatShow_end;
            }
            if (event.event == TIPC_WITHDRAWN)
            {
                break;
            }
        }
        printf ("\nSimulation resumed\n");
    }

    /* Release print lock */

    semGive (semSyncLock);
}

multi_printf ("Status display task deleted\n");
simStatShow_end:
close (sockfd_w);
tidShowSim = (int)NULL;
}

/*****
 *
 * simItem - simulated item
 *
 * This routine is the body of an item task.
 *
 * RETURNS: OK or ERROR
 *
 *****/
```

```
*/  
  
STATUS simItem  
(  
    int itemID                      /* item identifier */  
)  
{  
    int    sockfd_l;                /* descriptor for listening socket */  
    int    sockfd_s;                /* descriptor for server socket */  
    struct sockaddr_tipc addr;      /* socket address */  
    int    addrlen;                 /* socket address length */  
    char    inMsg[MSG_SIZE_MAX];    /* request msg from customer */  
    char    outMsg[MSG_SIZE_MAX];   /* reply msg to customer */  
    int    msgSize;                 /* message size (in bytes) */  
    uint    zone;                   /* zone ID of own node */  
    uint    cluster;                /* cluster ID of own node */  
    uint    node;                   /* node ID of own node */  
    char    itemName[8];            /* item name as character string */  
    int    haveItem;                /* TRUE if item has not been bought */  
    int    res;                     /* operation success indicator */  
    int    addr_size=sizeof(struct sockaddr_tipc); /* size of tipc sockaddr */  
  
    sprintf (itemName, "Item %d", itemID);  
    multi_printf ("%s created\n", itemName);  
  
    /* Create listening socket for item */  
  
    sockfd_l = socket (AF_TIPC, SOCK_SEQPACKET, 0);  
    if (sockfd_l < 0)  
    {  
        multi_printf ("    ERROR: %s can't create socket\n", itemName);  
        simErrors++;  
        if (errno == EAFNOSUPPORT)  
        {  
            multi_printf  
                (">>>>> MAKE SURE THAT TIPC IS ENABLED!!! <<<<<\n");  
        }  
        else  
            multi_printf("errno = %d (simErrors=%d).\n", errno, simErrors);  
        return ERROR;  
    }  
  
    res = listen (sockfd_l, 5);  
    if (res < 0)  
    {  
        multi_printf ("    ERROR: %s can't listen for customers\n", itemName);  
        simErrors++;  
        close (sockfd_l);  
        return ERROR;  
    }  
  
    /* Determine own node address */  
  
    if (0 > getsockname(sockfd_l, (struct sockaddr *)&addr, &addr_size))  
    {  
        multi_printf("    ERROR: %s can't determine own address\n", itemName);
```

```
    simErrors++;
    close(sockfd_l);
    return ERROR;
}

zone = tipc_zone (addr.addr.id.node);
cluster = tipc_cluster (addr.addr.id.node);
node = tipc_node (addr.addr.id.node);

/* bind socket and publish name */

addr.family = AF_TIPC;
addr.addrtype = TIPC_ADDR_NAME;
addr.scope = TIPC_ZONE_SCOPE;
addr.addr.name.name.type = TIPC_SALES_DEMO_TYPE;
addr.addr.name.name.instance = itemID;
addr.addr.name.domain = 0;
addrlen = sizeof (addr);

res = bind (sockfd_l, (struct sockaddr *)&addr, addrlen);
if (res < 0)
{
    multi_printf ("    ERROR: %s can't publish its name\n", itemName);
    simErrors++;
    close (sockfd_l);
    return ERROR;
}

/* Continue until a customer takes the item */

itemCount[itemID - DEMO_ITEM_ID_MIN]++;
haveItem = TRUE;
while (haveItem)
{
    /* Wait for a customer to request item */

    sockfd_s = accept (sockfd_l, (struct sockaddr *)&addr, &addrlen);
    if (sockfd_s < 0)
    {
        multi_printf ("    ERROR: %s connection failure\n", itemName);
        simErrors++;
        break;
    }

    msgSize = recv (sockfd_s, inMsg, sizeof (inMsg), 0);
    if (msgSize <= 0)
    {
        /* Try again if customer fails to send request */
        multi_printf ("    WARNING: %s didn't get customer request\n",
            itemName);
        simWarnings++;
    }
    else
    {
        sprintf (outMsg, "<%d.%d.%d>%s", zone, cluster, node, inMsg);
```

```
msgSize = strlen (outMsg) + 1;

res = send (sockfd_s, outMsg, msgSize, 0);
if (res != msgSize)
{
    /* Try again if can't send reply back to customer */
    multi_printf ("    WARNING: %s couldn't reply to customer\n",
        itemName);
    simWarnings++;
}
else
{
    char * ptr;
    if ((ptr = strchr (inMsg, '[')) != NULL)
        *ptr = '\0';
    multi_printf ("%s given to customer from %s\n",
        itemName, inMsg);
    haveItem = FALSE;
}
}

close (sockfd_s);
}
itemCount[itemID - DEMO_ITEM_ID_MIN]--;
itemsSold[itemID - DEMO_ITEM_ID_MIN]++;

/* Indicate item is no longer available */

close (sockfd_l);
semGive (semItems);

return (haveItem) ? ERROR : OK;
}

/*****
 *
 * simItemGen - simulated item generator
 *
 * This routine is the mainline for the item generator task.
 *
 */

void simItemGen
(
    int  maxItemID                /* max item ID for items */
)
{
    int  itemID;                  /* item ID for next item */
    int  waitTime;                /* delay before creating next item */
    char taskName[11];           /* task name character string */

    multi_printf ("Item generator created\n");

    FOREVER
    {
```

```
/* Only create a new item if fewer than the maximum exist */

if ((semTake (semItems, WAIT_FOREVER) == ERROR) || (!simActive))
    break;

/* Create randomly selected item */

itemID = randomGet (DEMO_ITEM_ID_MIN, maxItemID);
sprintf (taskName, "item_%d", itemID);
taskSpawn(taskName, ITEM_TASK_PRI, 0, ITEM_STACK_SIZE,
          (FUNCPTR)simItem, itemID, 0, 0, 0, 0, 0, 0, 0, 0, 0);

/* Do null print so generator pauses here when simulation is halted */

multi_printf ("");

/* Delay a bit to avoid creating items too quickly */

waitTime = randomGet (NEW_ITEM_WAIT_MIN, NEW_ITEM_WAIT_MAX);
taskDelay ((waitTime * ticksPerSec)/1000);
}

multi_printf ("Item generator deleted\n");
tidItemGen = (int)NULL;
}

/*****
 *
 * newItem - manually create simulated item
 *
 * This routine allows a user to create a specified item.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS newItem
(
    int itemID,                /* item identifier (0 = random) */
    int itemCount              /* # times to repeat (< 0 = forever) */
)
{
    int i;                    /* loop counter */
    int res = ERROR;          /* return code */

    if (!simActive)
    {
        printf ("Simulation not active on this node\n");
        return ERROR;
    }

    if (itemID == 0)
    {
        printf ("Random item(s) will be created\n");
    }
    else if ((itemID < DEMO_ITEM_ID_MIN) || (itemID > DEMO_ITEM_ID_MAX))
```

```
{
printf ("Item number must be in the range %d to %d\n",
        DEMO_ITEM_ID_MIN, DEMO_ITEM_ID_MAX);
return ERROR;
}

if (itemCount == 0)
{
    itemCount = 1;
}

taskPrioritySet (taskIdSelf (), ITEM_TASK_PRI);
for (i = 1; (itemCount < 0) || (i <= itemCount); i++)
{
    res = semTake (semItems, WAIT_FOREVER);
    if (res == ERROR)
    {
        printf ("Unable to create item\n");
        break;
    }

    res = simItem ((itemID > 0) ? itemID :
                  randomGet (DEMO_ITEM_ID_MIN, DEMO_ITEM_ID_MAX));

    /* don't exit if item had problems (some errors aren't fatal) */

}
return res;
}

/*****
 *
 * simCust - simulated customer
 *
 * This routine is the body of a customer task.
 *
 * NOTE: Currently no attempt is made to reduce the customer's maximum waiting
 * time following a failed attempt to obtain an item; instead, the customer
 * just starts waiting all over again. [Perhaps the appearance of the desired
 * item in the network (even though it wasn't obtained) is enough encouragement
 * to keep the customer hanging around longer than originally intended. :-)]
 *
 * The searchDomain parameter is normally passed as 0 to indicate that the
 * desired item can be anywhere in the network. A non-zero value (such as
 * the local node address) is used in simulation termination to connect to
 * local items.
 *
 * RETURNS: TRUE if desired item was obtained, otherwise FALSE
 *
 */

int simCust
(
    int    customerID,          /* unique ID for customer */
    int    itemID,              /* desired item */
    int    waitTime,            /* max time to wait (in ms) */

```



```

uint    searchDomain          /* where in network to look for item */
)
{
    int    sockfd_c;           /* descriptor for customer socket */
    int    sockfd_s;           /* descriptor for subscription socket */
    struct tipc_subscr subscr;  /* blocking subscription info */
    struct tipc_event event;    /* topology events (subscription) */
    struct sockaddr_tipc addr;  /* socket address */
    int    addrlen;            /* socket address length */
    char    msg[MSG_SIZE_MAX]; /* message to/from item */
    int    msgSize;            /* size of message (in bytes) */
    uint    zone;              /* zone ID of own node */
    uint    cluster;           /* cluster ID of own node */
    uint    node;              /* node ID of own node */
    char    custName[20];       /* customer name as character string */
    int    transactionID;       /* counts attempts to get item */
    int    needItem;           /* TRUE if customer still needs item */
    fd_set readFds;            /* socket to watch for item's reply */
    struct timeval timeLimit;    /* time to wait for item's reply */
    int    res;                /* operation success indicator */

    sprintf (custName, "Customer %d", customerID);
    multi_printf ("%s wants item %d within %d ms\n",
        custName, itemID, waitTime);

    /* subscribe to item name */
    subscr.seq.type = TIPC_SALES_DEMO_TYPE;
    subscr.seq.lower = itemID;
    subscr.seq.upper = itemID;
    subscr.timeout = waitTime;
    subscr.filter = TIPC_SUB_PORTS;

    sockfd_s = socket(AF_TIPC, SOCK_SEQPACKET, 0);
    if (sockfd_s < 0)
    {
        multi_printf ("cust: can't create subscr socket\n");
        simErrors++;
        if (errno == EAFNOSUPPORT)
        {
            multi_printf
                (">>>> MAKE SURE THAT TIPC IS ENABLED!!! <<<<\n");
        }
        else
            multi_printf("errno = %d (simErrors=%d).\n", errno, simErrors);
        return 0;
    }

    /* Determine own node address */
    addrlen = sizeof (struct sockaddr_tipc);
    if (getsockname(sockfd_s, (struct sockaddr *)&addr, &addrlen) < 0)
    {
        multi_printf ("    ERROR: %s can't determine own address\n",
            custName);
        simErrors++;
        close(sockfd_s);
        return 0;
    }

```

```
    }
    zone = tipc_zone(addr.addr.id.node);
    cluster = tipc_cluster(addr.addr.id.node);
    node = tipc_node(addr.addr.id.node);

    /* set up addressing */
    memset(&addr, 0, sizeof(addr));
    addr.family = AF_TIPC;
    addr.addrtype = TIPC_ADDR_NAME;
    addr.addr.name.name.type = TIPC_TOP_SRV;
    addr.addr.name.name.instance = TIPC_TOP_SRV;

    /* send subscription to topology server */
    if (sendto(sockfd_s, (char *)&subscr, sizeof (subscr), 0,
        (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        multi_printf ("simCust: can't connect to TOP server\n");
        simErrors++;
        close (sockfd_s);
        return 0;
    }

    customerCount[itemID - DEMO_ITEM_ID_MIN]++;
    needItem = TRUE;
    transactionID = 0;
    while (needItem)
    {
        /* Wait for desired item to appear */

        if (recv (sockfd_s, (char *)&event, sizeof (event), 0) != sizeof (event))
        {
            multi_printf("  ERROR: %s subscription failure\n",
                custName);
            simErrors++;
            break;
        }
        printf("simCust (%s): received an event\n", custName); /* ELMER */
        if (event.event == TIPC_SUBSCR_TIMEOUT)
        {
            multi_printf ("%s left without item %d after %d ms\n",
                custName, itemID, waitTime);
            customerExits++;
            break;
        }
        if (event.event == TIPC_WITHDRAWN)
        {
            /* ignore withdrawl events */
            continue;
        }

        /* termination of local simulator if searchDomain != 0;
        * clean up all remaining item tasks on local node */

        if ((searchDomain != 0) && (event.port.node != searchDomain))
        {
            /* ignore all non-local items */

```

```
        continue;
    }

    /* Create customer socket */

    sockfd_c = socket(AF_TIPC, SOCK_SEQPACKET, 0);
    if (sockfd_c < 0)
    {
        multi_printf("    ERROR: %s can't create socket\n", custName);
        simErrors++;
        break;
    }

    /* Try to choose item (using specified port id) */

    addr.family = AF_TIPC;
    addr.addrtype = TIPC_ADDR_ID;
    addr.addr.id = event.port;
    addrlen = sizeof(addr);

    sprintf (msg, "<%.%.%.%>[%.%.%]", zone, cluster, node,
            customerID, ++transactionID);
    msgSize = strlen (msg) + 1;

    res = sendto (sockfd_c, msg, msgSize, 0,
        (struct sockaddr *)&addr, addrlen);
    if (res != msgSize)
    {
        multi_printf (
            "    WARNING: %s unable to send request to item %d\n",
            custName, itemID);
        simWarnings++;
        customerRetries++;
    }
    else
    {
        FD_ZERO (&readFds);
        FD_SET (sockfd_c, &readFds);
        timeLimit.tv_sec = (waitTime + 999) / 1000;
        timeLimit.tv_usec = 0;
        if (select (sockfd_c + 1, &readFds, NULL, NULL, &timeLimit) == 0)
        {
            multi_printf (
                "    WARNING: %s missed item %d (no reply from item)\n",
                custName, itemID);
            simWarnings++;
            customerRetries++;
        }
        else
        {
            msgSize = recv (sockfd_c, msg, sizeof (msg), 0);
            if (msgSize <= 0)
            {
                multi_printf ("%s missed item %d (item rejected request)\n",
                    custName, itemID);
                customerRetries++;
            }
        }
    }
}
```

```

    }
    else
    {
        char * ptr;
        if ((ptr = strchr (msg, '>')) != NULL)
            *(ptr + 1) = '\0';
        multi_printf ("%s got item %d from %s\n",
            custName, itemID, msg);
        needItem = FALSE;
        customerSales++;
    }
}

close (sockfd_c);

/* Give TIPC 0.5s to withdraw item name in case item was on slow CPU */

if (needItem)
    taskDelay ((500 * ticksPerSec)/1000);
}
close(sockfd_s);
customerCount[itemID - DEMO_ITEM_ID_MIN]--;

/* Desired item obtained or unavailable, so exit */

semGive (semCustomers);

return needItem == FALSE;
}

/*****
 *
 * simCustGen - simulated customer generator
 *
 * This routine is the mainline for the customer generator task.
 *
 */

void simCustGen
(
    int  maxItemID                /* max item ID for customers */
)
{
    int  customerID;              /* unique ID for next customer */
    int  itemID;                  /* item ID for next customer */
    int  waitTime;                /* time to wait (in ms) */
    char taskName[11];            /* task name character string */

    multi_printf ("Customer generator created\n");

    customerID = 1;
    FOREVER
    {

        /* Only create a new customer if fewer than the maximum exist */

```

```

if ((semTake (semCustomers, WAIT_FOREVER) == ERROR) || (!simActive))
    break;

/* Create customer for randomly selected item */

itemID = randomGet (DEMO_ITEM_ID_MIN, maxItemID);
waitTime = randomGet (CUSTOMER_WAIT_MIN, CUSTOMER_WAIT_MAX);
sprintf (taskName, "cust%d_%d", customerID, itemID);
taskSpawn(taskName, CUSTOMER_TASK_PRI, 0, CUSTOMER_STACK_SIZE,
          (FUNCPTR)simCust, customerID, itemID, waitTime,
          0, 0, 0, 0, 0, 0, 0);

/* Increment customer ID (but don't let it exceed 3 digits) */

if (++customerID > 999)
    customerID = 1;

/* Do null print so generator stops here when simulation is halted */

multi_printf ("");

/* Delay a bit to avoid creating customers too quickly */

waitTime = randomGet (NEW_CUST_WAIT_MIN, NEW_CUST_WAIT_MAX);
taskDelay ((waitTime * ticksPerSec)/1000);
}

multi_printf ("Customer generator deleted\n");
tidCustGen = (int)NULL;
}

/*****
 *
 * newCust - manually create simulated customer
 *
 * This routine allows a user to create a customer looking for a specified item.
 * The customer is automatically assigned customer ID 1000 and a random time
 * to wait for the item.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS newCust
(
    int itemID,                /* desired item (0 = random) */
    int custCount              /* # times to repeat (< 0 = forever) */
)
{
    int i;                    /* loop counter */
    int res = ERROR;          /* return code */

    if (!simActive)
    {
        printf ("Simulation not active on this node\n");
    }

```

```
        return ERROR;
    }

    if (itemID == 0)
    {
        printf ("Random item(s) will be chosen for purchase\n");
    }
    else if ((itemID < DEMO_ITEM_ID_MIN) || (itemID > DEMO_ITEM_ID_MAX))
    {
        printf ("Item number must be in the range %d to %d\n",
            DEMO_ITEM_ID_MIN, DEMO_ITEM_ID_MAX);
        return ERROR;
    }

    if (custCount == 0)
    {
        custCount = 1;
    }

    taskPrioritySet (taskIdSelf (), CUSTOMER_TASK_PRI);
    for (i = 1; (custCount < 0) || (i <= custCount); i++)
    {
        res = semTake (semCustomers, WAIT_FOREVER);
        if (res == ERROR)
        {
            printf ("Unable to create customer\n");
            break;
        }

        res = simCust (1000, (itemID > 0) ? itemID :
            randomGet (DEMO_ITEM_ID_MIN, DEMO_ITEM_ID_MAX),
            randomGet (CUSTOMER_WAIT_MIN, CUSTOMER_WAIT_MAX), 0);

        /* don't exit if customer didn't get item (timeouts aren't fatal) */

    }
    return res;
}

/*****
 *
 * newSim - create sales demo
 *
 * This routine starts up the sales demo.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS newSim
(
    int maxItemID                /* max item ID for generated things */
)
{
    int i;                       /* loop counter */

```

```
if (simActive)
{
    multi_printf ("Simulation already active on this node\n");
    return ERROR;
}

if (maxItemID < 0)
{
    /* Won't auto-generate any items or customers */
}
else if (maxItemID == 0)
{
    maxItemID = DEMO_ITEM_ID_MAX;
}
else if ((maxItemID < DEMO_ITEM_ID_MIN) || (maxItemID > DEMO_ITEM_ID_MAX))
{
    printf ("Maximum item number must be in the range %d to %d\n",
            DEMO_ITEM_ID_MIN, DEMO_ITEM_ID_MAX);
    return ERROR;
}

/* Initialize data structures used by simulation */

ticksPerSec = sysClkRateGet ();
srand ((unsigned int)tickGet ());

semSyncLock = semBCreate (SEM_Q_FIFO, SEM_FULL);
semItems = semCCreate (SEM_Q_FIFO, MAX_ITEMS);
semCustomers = semCCreate (SEM_Q_FIFO, MAX_CUSTOMERS);

for (i = 0; i < NUM_DEMO_ITEMS; i++)
{
    itemsSold[i] = 0;
    itemCount[i] = 0;
    customerCount[i] = 0;
}
customerSales = 0;
customerExits = 0;
customerRetries = 0;

simErrors = 0;
simWarnings = 0;
simActive = TRUE;

/* Display initial status */

printf ("Sales demo created\n");

showSim ();

/* Spawn status display task, and optional customer and item generators */

tidShowSim = taskSpawn(
    "sim_show", SIM_STATUS_TASK_PRI, 0, SIM_STATUS_STACK_SIZE,
    (FUNCPTR)simStatShow, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
if (maxItemID >= 0)
```

```
{
    tidItemGen = taskSpawn(
        "item_gen", ITEM_GEN_TASK_PRI, 0, ITEM_GEN_STACK_SIZE,
        (FUNCPTR)simItemGen, maxItemID, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    tidCustGen = taskSpawn(
        "cust_gen", CUSTOMER_GEN_TASK_PRI, 0, CUSTOMER_GEN_STACK_SIZE,
        (FUNCPTR)simCustGen, maxItemID, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}
else
{
    tidItemGen = (int)NULL;
    tidCustGen = (int)NULL;
}

return OK;
}

/*****
 *
 * stopSim - halt sales demo
 *
 * This routine temporarily suspends the sales demo by creating an item 0.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS stopSim (void)
{
    struct sockaddr_tipc addr;          /* socket address */
    int addrlen;                       /* socket address length */
    int res;                           /* operation success indicator */

    if (!simActive)
    {
        printf ("Simulation not active on this node\n");
        return ERROR;
    }

    if (item0_fd >= 0)
    {
        printf ("Simulation already stopped by this node\n");
        return ERROR;
    }

    item0_fd = socket (AF_TIPC, SOCK_SEQPACKET, 0);
    if (item0_fd < 0)
    {
        multi_printf ("Unable to create socket for item 0\n");
        return ERROR;
    }

    addr.family = AF_TIPC;
    addr.addrtype = TIPC_ADDR_NAME;
    addr.scope = TIPC_ZONE_SCOPE;
    addr.addr.name.name.type = TIPC_SALES_DEMO_TYPE;
```



```
addr.addr.name.name.instance = 0;
addr.addr.name.domain = 0;
addrlen = sizeof (addr);

res = bind (item0_fd, (struct sockaddr *)&addr, addrlen);
if (res < 0)
{
    multi_printf ("Unable to publish name for item 0\n");
    close (item0_fd);
    item0_fd = -1;
    return ERROR;
}

return OK;
}

/*****
 *
 * startSim - resume sales demo
 *
 * This routine resumes the sales demo by destroying item 0.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS startSim (void)
{
    if (!simActive)
    {
        printf ("Simulation not active on this node\n");
        return ERROR;
    }

    if (item0_fd < 0)
    {
        printf ("Simulation not stopped by this node\n");
        return ERROR;
    }

    close (item0_fd);
    item0_fd = -1;

    return OK;
}

/*****
 *
 * simDie - terminate sales demo
 *
 * This routine terminates the sales demo.
 *
 * NOTE: This routine must be run at "customer" task priority so that each
 * item it consumes has a chance to close its socket (& unbind the associated
 * item name) before the the termination code activates another customer --
 * otherwise the new customer may try to grab the just deleted item!
 */
```

```
*
* RETURNS: OK or ERROR
*
*/

LOCAL STATUS simDie (void)
{
    struct tipc_subscr subscr;          /* general purpose subscription info */
    struct tipc_event event;           /* subscription event info */
    struct sockaddr_tipc addr;         /* topology server socket address */
    struct sockaddr_tipc topsrv;       /* topology server socket address */
    int sockfd_c;                     /* socket descriptor for customer 0 */
    int itemID;                       /* loop counter for processing items */
    int addr_size=sizeof (struct sockaddr_tipc); /* size of tipc sockaddr */
    int my_node;                      /* node ID of this node */

    /* Tell generator and status display tasks to shut down */

    simActive = FALSE;
    semGive (semItems);
    semGive (semCustomers);

    /* Wait for customer tasks to timeout & terminate on their own */

    for (itemID = 0; itemID < NUM_DEMO_ITEMS; itemID++)
        while (customerCount[itemID] > 0)
            taskDelay (1);

    /* Wait for generator and status display tasks to terminate */

    while (tidItemGen != (int)NULL)
        taskDelay (1);
    while (tidCustGen != (int)NULL)
        taskDelay (1);
    while (tidShowSim != (int)NULL)
        taskDelay (1);

    /* Display status before consuming unsold items */

    showSim ();

    /* Consume unsold items (from this CPU only!) by creating fake customers */

    sockfd_c = socket (AF_TIPC, SOCK_SEQPACKET, 0);
    if (sockfd_c < 0)
    {
        multi_printf ("ERROR: can't create socket\n");
        goto simDie_end;
    }

    memset(&topsrv, 0, sizeof(topsrv));
    topsrv.family = AF_TIPC;
    topsrv.addrtype = TIPC_ADDR_NAME;
    topsrv.addr.name.name.type = TIPC_TOP_SRV;
    topsrv.addr.name.name.instance = TIPC_TOP_SRV;
```

```
subscr.seq.type = TIPC_SALES_DEMO_TYPE;
subscr.seq.lower = DEMO_ITEM_ID_MIN;
subscr.seq.upper = DEMO_ITEM_ID_MAX;
subscr.timeout = 10; /* need a nominal delay */
subscr.filter = TIPC_SUB_PORTS;

if (0 > getsockname(sockfd_c, (struct sockaddr *)&addr, &addr_size))
{
    multi_printf("ERROR: can't determine own address\n");
    goto simDie_closeend;
}
my_node = addr.addr.id.node;

if (sendto (sockfd_c, (char *)&subscr, sizeof (subscr), 0,
            (struct sockaddr *)&topsrv, sizeof (topsrv)) < 0)
{
    /* note that topsrv is set */
    multi_printf("ERROR: subscription failure\n");
    goto simDie_closeend;
}

FOREVER
{
    if (recv (sockfd_c, (char *)&event, sizeof (event), 0) < 0)
    {
        /* FIXME: some items may be left over after exiting */
        multi_printf("ERROR: receive problem\n");
        goto simDie_closeend;
    }
    if (event.event == TIPC_SUBSCR_TIMEOUT)
        break; /* all done */

    if ((event.event == TIPC_PUBLISHED) &&
        (event.port.node == my_node))
    {
        simCust(0, event.found_lower, 100, my_node); /* stay on card */
    }
    else
    {
        /* ignore event */
    }
}

simDie_closeend:
close(sockfd_c);

simDie_end:
/* Display final status & exit demo */

showSim ();

semDelete (semCustomers);
semDelete (semItems);
semDelete (semSyncLock);

printf ("Sales demo terminated\n");
```

```
    return OK;
}

/*****
 *
 * killSim - initiate sales demo termination
 *
 * This routine spawns a task to terminate the sales demo.
 *
 * Note: The deletion is done by a separate task for several reasons:
 * 1) It avoids locking up the VxWorks shell task during the termination
 *    phase, which can last for a number of seconds.
 * 2) It provides a simple way to ensure the termination code runs at the
 *    proper task priority level (i.e. don't have to save & restore the current
 *    task priority of the VxWorks shell task).
 * 3) It masks a minor side-effect of the VxWorks select() routine which can
 *    set errno to a non-zero value even when everything is working properly.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS killSim (void)
{
    if (!simActive)
    {
        printf ("Simulation not active on this node\n");
        return ERROR;
    }

    if (taskSpawn(
        "sim_kill", CUSTOMER_TASK_PRI, 0, CUSTOMER_STACK_SIZE,
        (FUNCPTR)simDie, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
    {
        printf ("Unable to terminate simulation\n");
        return ERROR;
    }

    return OK;
}

/*****
 *
 * memSim - show memory usage during simulation
 *
 * This routine pauses the simulation for a memory dump.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS memSim (void)
{
    if (!simActive)
    {
        printf ("Simulation not active on this node\n");
    }
}
```

```
        return ERROR;
    }

    stopSim ();
    memShow (1);
    startSim ();
    return OK;
}
```


F

TIPC Log Messages

[F.1 Introduction 247](#)

[F.2 Log Messages 248](#)

F.1 Introduction

This appendix lists the messages that can appear in the TIPC log. There are three types of messages:

- Informational (**TIPC info**)
- Warnings (**TIPC warning**)
- Errors (**TIPC error**)

F.2 Log Messages

The following messages can appear in the TIPC log:

TIPC info Messages

```
TIPC info: Enabled bearer <%s>, discovery domain %s, priority %u
TIPC info: Blocking bearer <%s>
TIPC info: Disabling bearer <%s>
TIPC info: Activated (version " TIPC_MOD_VER ")
TIPC info: tid %x take %x (%d)
TIPC info: tid %x free %x
TIPC info: Bearer %s is down
TIPC info: Bearer %s is up
TIPC info: tTipcTask terminated normally
TIPC info: Resetting link <%s>, requested by peer
TIPC info: Resetting link <%s>, requested by peer "
TIPC info: Resetting link <%s>, changeover initiated by peer
TIPC info: Reception queue empty
TIPC info: Contents of Reception queue:
TIPC info: buffer %x invalid
TIPC info: Contents of unsent queue:
TIPC info: Contents of send queue:
TIPC info: Empty send queue
TIPC info: Started in network mode
TIPC info: Own node address %s, network identity %u
TIPC info: Left network mode
TIPC info: Established link <%s> on network plane %c
TIPC info: New link <%s> becomes standby
TIPC info: Old link <%s> becomes standby
TIPC info: Old link <%s> becomes standby
TIPC info: Lost standby link <%s> on network plane %c
TIPC info: Lost link <%s> on network plane %c
TIPC info: Lost link <%s> on network plane %c
TIPC info: Lost contact with %s
```

TIPC warning Messages

```
TIPC warning: Multicast link creation failed, no memory
TIPC warning: Incomplete multicast delivery, no memory
TIPC warning: Media <%s> rejected
TIPC warning: Bearer <%s> rejected, not supported in standalone mode
TIPC warning: Bearer <%s> rejected, illegal name
TIPC warning: Bearer <%s> rejected, illegal discovery domain
TIPC warning: Bearer <%s> rejected, illegal priority
TIPC warning: Bearer <%s> rejected, media <%s> not registered
TIPC warning: Bearer <%s> rejected, already enabled
TIPC warning: Bearer <%s> rejected, duplicate priority
TIPC warning: Bearer <%s> priority adjustment required %u->%u
TIPC warning: Bearer <%s> rejected, bearer limit reached (%u)
```


TIPC warning: Bearer <%s> rejected, enable failure (%d)
TIPC warning: Attempt to block unknown bearer <%s>
TIPC warning: Attempt to disable unknown bearer <%s>
TIPC warning: Memory squeeze; dropped remote link subscription
TIPC warning: Invalid configuration message discarded
TIPC warning: Duplicate %s using %s seen on <%s>
TIPC warning: Memory squeeze; Failed to create node
TIPC warning: Ignoring request for second link to node %s
TIPC warning: Memory squeeze; Failed to create link
TIPC warning: Out of buffers; incoming SM message discarded
TIPC warning: Can't duplicate buffer to send over Ethernet! (errno=0x%08x)
TIPC warning: Out of buffers; incoming Ethernet message discarded
TIPC warning: Interface %s not found
TIPC warning: Cannot enable more than %d Ethernet Bearers.
TIPC warning: Unable to bind to interface %s
TIPC warning: Attempt to re-initialize Ethernet media ignored
TIPC warning: Cannot start ethernet media
TIPC warning: Link creation failed, no memory
TIPC warning: Link creation failed, no memory for print buffer
TIPC warning: Resetting link <%s>, peer not responding
TIPC warning: Resetting link <%s>, send queue full", l_ptr->name);
TIPC warning: Resetting all links to %s
TIPC warning: Resetting link <%s>
TIPC warning: Retransmission failure on link <%s>
TIPC warning: Resetting link <%s>
TIPC warning: Resetting link <%s>, priority change %u->%u
TIPC warning: Link changeover error, tunnel link no longer available
TIPC warning: Link changeover error, unable to send tunnel msg
TIPC warning: Link changeover error, peer did not permit changeover
TIPC warning: Link changeover error, unable to send changeover message
TIPC warning: Link changeover error, unable to send duplicate msg
TIPC warning: Link changeover error, duplicate msg dropped
TIPC warning: Link switchover error, got too many tunnelled messages
TIPC warning: Link changeover error, original msg dropped
TIPC warning: Link unable to unbundle message(s)
TIPC warning: Link unable to fragment message
TIPC warning: Link unable to reassemble fragmented message
TIPC warning: Bulk publication failure
TIPC warning: Bulk publication not sent
TIPC warning: Bulk publication not sent
TIPC warning: Memory squeeze; failed to distribute publication
TIPC warning: Publication distribution to cluster failed
TIPC warning: Publication distribution to cluster failed
TIPC warning: Publication distribution to cluster failed
TIPC warning: Memory squeeze; failed to distribute name table msg
TIPC warning: Failed to distribute name table msg
TIPC warning: Memory squeeze; failed to distribute name table msg
TIPC warning: Failed to distribute name table msg
TIPC warning: Invalid name table item received
TIPC warning: Bulk route publication failure
TIPC warning: Bulk route update not sent
TIPC warning: Memory squeeze; failed to distribute route
TIPC warning: Route distribution to cluster failed
TIPC warning: Memory squeeze; failed to distribute route msg
TIPC warning: Failed to distribute route msg
TIPC warning: Memory squeeze; failed to distribute route msg

TIPC warning: Failed to distribute route msg
TIPC warning: Invalid routing table item received
TIPC warning: Publication creation failure, no memory
TIPC warning: Name sequence creation failed, no memory
TIPC warning: Cannot publish {%,%,%}, overlap error
TIPC warning: Cannot publish {%,%,%}, overlap error
TIPC warning: Cannot publish {%,%,%}, no memory
TIPC warning: Failed to publish illegal {%,%,%}
TIPC warning: Failed to publish reserved name <%,%,%>
TIPC warning: Publication failed, local publication limit reached (%u)
TIPC warning: Failed to create subscription for {%,%,%}
TIPC warning: Publication failed, local publication limit reached (%u)
TIPC warning: Could not add element %x (max %u allowed for this type)
TIPC warning: Memory squeeze; unable to record new region
TIPC warning: Node creation failed, no memory
TIPC warning: Unable to deliver multicast message(s)
TIPC warning: Port creation failed, no memory
TIPC warning: Port creation failed, reference table exhausted
TIPC warning: Port creation failed, no memory
TIPC warning: tipcSm (%s): %d %s unavailable in the last %d system ticks
TIPC warning: tipcSm (%s): Out of %s, incoming SM message discarded (tick#: %d)
TIPC warning: Shared memory bearer already enabled
TIPC warning: Shared memory bearer '%s' not a valid name
TIPC warning: Attempt to re-initialize Shared Memory media ignored
TIPC warning: Cannot start shared memory media
TIPC warning: Subscription rejected, subscription limit reached (%u)
TIPC warning: Subscription rejected, no memory
TIPC warning: Subscription rejected, illegal request
TIPC warning: Subscriber rejected, invalid subscription size
TIPC warning: Subscriber rejected, no memory
TIPC warning: Subscriber rejected, unable to create port

TIPC error Messages

TIPC error: Unable to create configuration service
TIPC error: Unable to create configuration service identifier
TIPC error: Cannot start network communication
TIPC error: Unable to initiate network communication
TIPC error: Unable to allocate additional signals
TIPC error: Unable to allocate more signal entries
TIPC error: Attempt to delete non-existent link
TIPC error: Unknown link event %u in WW state
TIPC error: Unknown link event %u in WU state
TIPC error: Unknown link event %u in RU state
TIPC error: Unknown link event %u in RR state
TIPC error: Unknown link state %u/%u
TIPC error: Unexpected changeover message on link <%s>
TIPC error: Unable to de-list cluster publication
TIPC error: Unable to de-list node publication
TIPC error: Unable to remove local publication
TIPC error: tipc_nametbl_stop(): hash chain %u is non-null
TIPC error: tipc_routetbl_stop(): routing table has %u entries
TIPC error: Unable to remove local route

```
TIPC error: Attempt to create third link to %s
TIPC error: Attempt to establish second link on <%s> to %s
TIPC error: Attempt to acquire reference to non-existent object
TIPC error: Reference table not found during acquisition attempt
TIPC error: Reference table not found during discard attempt
TIPC error: Attempt to discard reference to non-existent object
TIPC error: Attempt to discard non-existent reference
TIPC error: tipc_ref_unlock() invoked using invalid reference
TIPC error: Unable to allocate additional buffers
TIPC error: vxskb_tuple_get(): pCluster == NULL
TIPC error: vxskb_tuple_get(): newClBlk == NULL
TIPC error: vxskb_tuple_get(): netClBlkJoin == NULL
TIPC error: vxskb_tuple_get(): mBlkIdNew == NULL
TIPC error: vxskb_tuple_get(): mBlkIdNew == NULL
TIPC error: Unable to allocate %d byte buffer
TIPC error: Failed to create subscription service
```


Index

A

- a (addr) (tipcConfig command option) 65
- a (address) (parameter in configuration string) 48
- accept() 10
 - see also* Appendix B: Routines

B

- b (tipcConfig command option) 65
- bd (tipcConfig command option) 65
- be (bearer)
 - parameter in configuration string 51
 - tipcConfig command option 75–77
- bootline configuration (build component) 20, 47
 - in a Workbench build 58
- BSPs
 - support for DSHM 35
 - support for shared memory 33
- build components 20–27
 - bootline configuration 20, 47
 - in a Workbench build 58
 - Build TIPC from object library 21
 - Build TIPC from source 22
 - DSHM Primary Interface 24
 - Ethernet 23
 - INCLUDE_BUILD_TIPC_SRC 22
 - INCLUDE_CONFIG_TIPC_SOCKET_API 26

- INCLUDE_DSHM_SVC_TIPC_PRIM 24
- INCLUDE_TIPC 20
- INCLUDE_TIPC_CONFIG_HOOK_BOOT 20, 47
- INCLUDE_TIPC_CONFIG_HOOK_USER 21, 47
- INCLUDE_TIPC_CONFIG_STR 24, 46, 47
- INCLUDE_TIPC_DEFINES 26, 42
 - parameters, table of 43
- INCLUDE_TIPC_HEND_INIT 27, 45
- INCLUDE_TIPC_IP 22
- INCLUDE_TIPC_IS 27
- INCLUDE_TIPC_MEDIA_ETH 23, 33
- INCLUDE_TIPC_MEDIA_SM 23
- INCLUDE_TIPC_MEMPOOL 20, 31
- INCLUDE_TIPC_NOCFG_SERVICE 25
- INCLUDE_TIPC_NODEBUG 24
- INCLUDE_TIPC_NOSOCKET 25
- INCLUDE_TIPC_NOSYS_MSGS 24
- INCLUDE_TIPC_ONLY 23, 28, 37
 - debugging with WDB target agent 38
- INCLUDE_TIPC_SHOW 24, 42
- INCLUDE_TIPC_TS 27
- INCLUDE_USE_LIBTIPC 21
- INCLUDE_WVTIPC 27
- No TIPC configuration 25
- No TIPC debug 24
- No TIPC socket API 25
- No TIPC system messages 24
- Shared Memory 23

- table of build components 20
- TIPC 20
- TIPC and IP network stacks present 22
- TIPC configuration and display routines 24, 42
- TIPC instrumentation 27
- TIPC inventory simulation demo 27
- TIPC memory pool 20, 31–32
- TIPC network stack
 - in a Workbench build 59
- TIPC network stack only 23, 28, 37
 - debugging with WDB target agent 38
- TIPC prioritized interfaces 27, 45
- TIPC socket API 26, 30–31
- TIPC static configuration 24, 46, 47, 58
- TIPC System Defines 26, 42
 - parameters, table of 43
- TIPC test suite demo 27
- user configuration 21, 47
 - in a Workbench build 58
- Build TIPC from object library (build component) 21
- Build TIPC from source (build component) 22
- building VxWorks to include TIPC 19–61
 - build components 20
 - see also, as main entry, build components*
 - Workbench build 55–61

C

- close() 92
 - see also* Appendix B: Routines
- clusters
 - defined 6
 - links between 78
 - links within 78
- components, *see* build components
- configuration string
 - a (address) parameter 48
 - be parameter 51
 - log parameter 49, 50
 - max_clusters parameter 49, 50
 - max_nodes parameter 49, 50
 - max_ports parameter 49, 50

- max_publ parameter 51
- max_subscr parameter 51
- max_zones parameter 51
- netid parameter 51
- configuring Wind River TIPC 46
 - configuration string 46
 - dynamic configuration 46
 - static configuration 46
- connect() 10
 - see also* Appendix B: Routines

D

- d (dest) (tipcConfig command option) 66, 81–82, 85
- debugging TIPC 38
- Default Network ID (static configuration parameter) 43
- distributed shared memory (DSHM), *see* DSHM
- domains 77–81
- DSHM 35
 - media type
 - supported BSPs 35
- DSHM Primary Interface (build component) 24
- dynamic configuration 46
 - Workbench build 58

E

- Ethernet 33
 - media type 8
- Ethernet (build component) 23

F

- footprint reduction 27–29

G

- getsockname() 13

see also Appendix B: Routines

H

h (help) (tipcConfig command option) 66
 header-file definitions 214
 HEND interfaces and drivers 45

I

i (tipcConfig command option) 66
 implied connection request 10
 importance level 11
 INCLUDE_BUILD_TIPC_SRC (build component) 22
 INCLUDE_CONFIG_TIPC_SOCKET_API (build component) 26
 parameters, table of 30
 INCLUDE_DSHM_SVC_TIPC_PRIM (build component) 24
 INCLUDE_TIPC (build component) 20
 INCLUDE_TIPC_CONFIG_HOOK_BOOT (build component) 20, 47
 INCLUDE_TIPC_CONFIG_HOOK_USER (build component) 21, 47
 INCLUDE_TIPC_CONFIG_STR (build component) 24, 46, 47
 INCLUDE_TIPC_DEFINES (build component) 26, 42
 INCLUDE_TIPC_HEND_INIT (build component) 27, 45
 INCLUDE_TIPC_IP (build component) 22
 INCLUDE_TIPC_IS (build component) 27
 INCLUDE_TIPC_MEDIA_ETH (build component) 23, 33
 INCLUDE_TIPC_MEDIA_SM
 shared-memory (build component) 23
 INCLUDE_TIPC_MEDIA_SM (build component)
 parameters, table of 34
 INCLUDE_TIPC_MEMPOOL (build component) 20, 31
 parameters, table of 31

INCLUDE_TIPC_NOCFG_SERVICE (build component) 25
 INCLUDE_TIPC_NODEBUG (build component) 24
 INCLUDE_TIPC_NOSOCKET (build component) 25
 INCLUDE_TIPC_NOSYS_MSGS (build component) 24
 INCLUDE_TIPC_ONLY (build component) 23, 28, 37
 debugging with WDB target agent 38
 INCLUDE_TIPC_SHOW (build component) 24, 42
 INCLUDE_TIPC_TS (build component) 27
 INCLUDE_USE_LIBTIPC (build component) 21
 INCLUDE_WVTIPC (build component) 27

L

l (tipcConfig command option) 66
 links
 between clusters 78
 between zones 80
 defined 6
 multiple, for load sharing and switchover 8
 within a cluster 78
 Linux TIPC
 comparison with Wind River TIPC 3
 interoperable with Wind River TIPC 1
 listen() 10
 see also Appendix B: Routines
 load sharing 8
 log
 parameter in configuration string 49, 50
 tipcConfig command option 67
 sample output 83
 lp (tipcConfig command option) 68
 ls (tipcConfig command option) 68
 sample output 83
 lsr (tipcConfig command option) 68
 lt (tipcConfig command option) 69
 lw (tipcConfig command option) 69

M

- m (tipcConfig command option) 69
- Max Clusters (static configuration parameter) 43
- Max Nodes (static configuration parameter) 43
- Max Ports (static configuration parameter) 43
- Max Publications (static configuration parameter) 44
- Max Remotes (static configuration parameter) 43
- Max Subscriptions (static configuration parameter) 44
- Max Zones (static configuration parameter) 44
- max_clusters
 - parameter in configuration string 49, 50
 - tipcConfig command option 69
- max_nodes
 - parameter in configuration string 49, 50
 - tipcConfig command option 70
- max_ports
 - configuration string parameter 49, 50
 - parameter in configuration string 49, 50
 - tipcConfig command option 70
- max_publ
 - parameter in configuration string 51
 - tipcConfig command option 70
- max_remotes (tipcConfig command option) 71
- max_subscr
 - parameter in configuration string 51
 - tipcConfig command option 71
- max_zones
 - parameter in configuration string 51
 - tipcConfig command option 71
- media types 32
 - DSHM 35
 - Ethernet 33
 - shared memory 33
- message, definition of 10
- mng (tipcConfig command option) 72, 84–85
- multicasting 17–18

N

- n (tipcConfig command option) 72
- netid

- parameter in configuration string 51
- tipcConfig command option 86
- network addresses 12
 - notation for 7
- No TIPC configuration (build component) 25
- No TIPC debug (build component) 24
- No TIPC socket API (build component) 25
- No TIPC system messages (build component) 24
- nt (tipcConfig command option) 73, 86–89
 - sample output 87

P

- p (ports) (tipcConfig command option) 73
 - sample output 89
- port address
 - functional 13
 - physical 13
- port name 14
 - type and name components 14
- port name sequence 14

R

- recv() 10
 - see also Appendix B: Routines
- recvfrom() 10
 - see also Appendix B: Routines
- recvmsg() 10
 - see also Appendix B: Routines
- remote management (tipcConfig utility) 84–85
- Remote Management (static configuration parameter) 44

S

- s (tipcConfig command option) 74
- secondary nodes not supported 6
- send() 10
 - see also Appendix B: Routines
- sendmsg() 93

see also Appendix B: Routines
 sendto() 10
see also Appendix B: Routines
 shared memory 33
 media type 8
 supported BSPs 33
 Shared Memory (build component) 23
 parameters, table of 34
 show routines
 tipcDataPoolShow 42
 tipcSysPoolShow 42
 sockaddr_tipc structure 3
 socket() 10
see also Appendix B: Routines
 static configuration 46
 Workbench build 58
 subscriptions 91–93
 creating 92
 receiving an event notification 93
 service provided by TIPC 18
 switchover 8
 System Viewer, *see* Wind River System Viewer

T

target agent, WDB 38
 target server
 starting 40
 test suite 107–113
 including in a project 108
 running tests 108
 sample output 111
 tests included 110
 tipcTC shell command 109
 tipcTS shell command 109
 TIPC (build component) 20
 TIPC and IP network stacks present (build component) 22
 TIPC configuration and display routines (build component) 24, 42
 TIPC instrumentation (build component) 27
 TIPC inventory simulation demo (build component) 27
 TIPC memory pool (build component) 20, 31–32
 parameters, table of 31
 TIPC network stack (build component)
 in a Workbench build 59
 TIPC network stack only (build component) 23, 28, 37
 debugging with WDB target agent 38
 TIPC prioritized interfaces (build component) 27, 45
 TIPC protocol 1
see also, Wind River TIPC
 addressing 12
 address resolution 15
 address types 12
 functional port addresses 13
 multiple port names or name sequences
 bound to a socket 15
 network addresses 12
 notation for addresses 7
 physical port addresses 13
 same port name or name sequence bound
 to multiple ports 15
 basic concepts 5–18
 clusters 6
 links 6
 messaging 9
 multicasting 17–18, ??–18
 network structure 6
 port name 14
 ports 9
 subscription service provided 18
 subscriptions 91–93
 creating 92
 receiving an event notification 93
 zones 6
 TIPC socket API (build component) 26, 30–31
 parameters, table of 30
 TIPC static configuration (build component) 24, 46, 47, 58
 TIPC System Defines (build component) 26, 42
 parameters, table of 43
 TIPC test suite demo (build component) 27
 tipc_addr() 13
see also Appendix B: Routines
 tipc_cluster() 13
see also Appendix B: Routines

TIPC_DEF_MAX_CLUSTERS (static configuration parameter) 43
TIPC_DEF_MAX_NODES (static configuration parameter) 43
TIPC_DEF_MAX_PORTS (static configuration parameter) 43
TIPC_DEF_MAX_PUBS (static configuration parameter) 44
TIPC_DEF_MAX_REMOTES (static configuration parameter) 43
TIPC_DEF_MAX_SUBS (static configuration parameter) 44
TIPC_DEF_MAX_ZONES (static configuration parameter) 44
TIPC_DEF_NET_ID (static configuration parameter) 43
TIPC_DEF_REMOTE_MGT (static configuration parameter) 44
tipc_node() 13
 see also Appendix B: Routines
tipc_zone() 13
 see also Appendix B: Routines
tipcConfig utility 63–89
 a (addr) command option 65
 b command option 65
 bd command option 65
 be (bearer) command option 75–77
 constraints on order of command options 75
 d (dest) command option 66, 81–82, 85
 h (help) command option 66
 i command option 66
 l command option 66
 log command option 67
 sample output 83
 lp command option 68
 ls command option 68
 sample output 83
 lsr command option 68
 lt command option 69
 lw command option 69
 m command option 69
 max_clusters command option 69
 max_nodes command option 70
 max_ports command option 70
 max_publ command option 70

 max_remotes command option 71
 max_subscr command option 71
 max_zones command option 71
 mng command option 72, 84–85
 n command option 72
 netid command option 86
 nt command option 73, 86–89
 sample output 87
 p (ports) command option 73
 sample output 89
 remote management 84–85
 s command option 74
 V command option 74
 v command option 74
tipcConfigInfoGet()
 for dynamic configuration 53
 with a command-line build 54
 with a Workbench build 58
tipcDataPoolShow show routine 42
tipcDataPoolShow() 42
 see also Appendix B: Routines
tipcSysPoolShow show routine 42
tipcSysPoolShow() 42
 see also Appendix B: Routines
Transparent Inter Process Communication (TIPC)
 protocol. *see* TIPC protocol

U

user configuration (build component) 21, 47
 in a Workbench build 58

V

V (tipcConfig command option) 74
v (tipcConfig command option) 74
VxWorks
 including Wind River TIPC in a build
 building from Workbench 55–61
VxWorks simulator 95–99
 simulating a network of TIPC nodes 96
 DSHM 99

- Ethernet 96
- shared memory 98
- simulating a standalone TIPC node 96

W

- WDB agent proxy 38
- WDB target agent 38
- Wind River System Viewer 101–105
 - event levels 102
 - including instrumentation for TIPC in a build 104
 - TIPC events covered 102
- Wind River TIPC
 - see also*, TIPC protocol
 - comparison with Linux version 3
 - configuration string
 - setting parameters in 48
 - table of parameters 48
 - configuring 46
 - configuration string 46
 - dynamic configuration 46
 - static configuration 46
 - connection types 10
 - domains 77–81
 - footprint reduction 27–29
 - header-file definitions 214
 - interoperability with other releases 3
 - interoperable with open-source Linux version 1
 - media types 32
 - message reliability 11
 - overview 2
 - rejected messages 11
 - secondary nodes not supported 6
 - supported media 8
 - system parameters
 - Default Network ID 43
 - Max Clusters 43
 - Max Nodes 43
 - Max Ports 43
 - Max Publications 44
 - Max Remotes 43
 - Max Subscriptions 44
 - Max Zones 44
 - Remote Management 44
 - TIPC_DEF_MAX_CLUSTERS 43
 - TIPC_DEF_MAX_NODES 43
 - TIPC_DEF_MAX_PORTS 43
 - TIPC_DEF_MAX_PUBS 44
 - TIPC_DEF_MAX_REMOTES 43
 - TIPC_DEF_MAX_SUBS 44
 - TIPC_DEF_MAX_ZONES 44
 - TIPC_DEF_NET_ID 43
 - TIPC_DEF_REMOTE_MGT 44
 - test suite 107–113
 - including in a project 108
 - running tests 108
 - sample output 111
 - tests included 110
 - tipcTC shell command 109
 - tipcTS shell command 109
 - tipcConfig utility, *see as main entry*, tipcConfig utility
 - undeliverable messages 11
- Wind River VxWorks Simulator, *see* VxWorks simulator
- Workbench
 - building VxWorks to include TIPC 55–61

Z

- zones
 - defined 6
 - links between 80