RealCT Direct API

Developer Guide

Document Number 934-010-82
Printed August 2001

Brooktrout Technology®

General Notices

Brooktrout Technology reserves the right to make improvements and/or changes in the products and programs described in this Developer Guide at any time without notice. Every attempt has been made to insure that the information contained in this document is accurate and complete. Brooktrout Technology will not be responsible for any inaccuracies or omissions in this or any of its other technical publications.

Printed in the United States of America.

Trademarks

International Notice

Due to differing national regulations and approval requirements, certain Brooktrout products are designed for use only in specific countries, and may not function properly in a country other than the country of designated use. As a user of these products, you are responsible for ensuring that the products are used only in the countries for which they were intended. For information on specific products, contact Brooktrout Technology.

151 Albright Way
Los Gatos, CA 95032-1801
408-370-0881
www.brooktrout.com

Brooktrout Technical Support

For Brooktrout Technical Support, see *Getting Help* on page xxi.

Limited Warranty

Brooktrout, Inc. (“Brooktrout”) warrants the hardware component of the product described in this documentation (the “Product”) to be free from defects in materials and workmanship under normal and proper use for a period of five years from the date of purchase from Brooktrout. Brooktrout also warrants the disk on which software and firmware are recorded to be free from defects in materials and workmanship under normal and proper use for a period of 90 days from the date of purchase from Brooktrout. **This warranty does not apply to the software and firmware themselves.** This warranty also does not apply to any expendable components, any damage resulting from abuse of the Product, or normal wear and tear. In the event of a warranty claim, the item, if in the opinion of Brooktrout it is proved to be defective, will be repaired or replaced with a functionally equivalent item, at Brooktrout's sole option, upon delivery to Brooktrout of the defective item, together with a dated proof of purchase and specification of the problem. Brooktrout is not responsible for transportation and related charges in connection with shipment of items to Brooktrout for warranty service. Brooktrout reserves the right to charge for inspection at Brooktrout's then prevailing rates of returned items if it is determined that the items were not defective within the terms of the warranty. To obtain warranty service return the Product, contact Brooktrout Technical Support.

With respect to software and firmware, it should be understood that these components are complex works which may contain undiscovered defects. Although the software and firmware provided with the Product contain substantially the features described in the documentation, to the extent applicable to the product purchased, Brooktrout does not warrant that the operation of such software and firmware will meet the user’s requirements or be uninterrupted or free of errors.

No oral or written information or advice given by Brooktrout or its authorized representatives will create a warranty or increase the scope of this warranty. No representative, agent, dealer or employee of Brooktrout is authorized to give any other warranty or to assume for Brooktrout any other liability in connection with the sale and service of the Product. **Except as expressly agreed by Brooktrout in writing, Brooktrout makes no representations or warranties of any kind, express or implied, with respect to the Product or any hardware, software or firmware components thereof. In particular, but without limitation of the foregoing, Brooktrout disclaims all implied warranties of merchantability or fitness for a particular purpose and there are no warranties that extend beyond the description or duration of this warranty.** Some states or countries do no allow the exclusion of implied warranties so the above exclusion may not apply to you.

In no event shall Brooktrout be liable for loss of profits or indirect, special, incidental, or consequential damages arising out of the use of or inability to use the Product. The sole and exclusive remedy, in contract, tort or otherwise, available for a breach of this warranty and for any and all claims arising out of or in any way connected with the purchase of the Product shall be limited to the repair or replacement of any defective item or, at Brooktrout's sole option, the payment of actual direct damages not to exceed the payments made to Brooktrout for the Product in question. Some states do not allow the limitation or exclusion of liability for incidental or consequential damages, so the above limitation and exclusion may not apply to you.

This warranty gives you specific legal rights. You may also have other rights which vary from state-to-state or country-to-country. Any provision of this warranty that is prohibited or unenforceable in any jurisdiction shall, as to such jurisdiction, be ineffective to the extent of such prohibition or unenforceability without invalidating the remaining provisions hereof or affecting the validity of enforceability of such provision in any other jurisdiction.

BEFORE USING THIS SOFTWARE, YOU SHOULD CAREFULLY READ THE FOLLOWING LICENSE AGREEMENT WHICH APPLIES TO THE SOFTWARE PRODUCT IN THE PACKAGE (THE “SOFTWARE”). USING THIS SOFTWARE INDICATES YOUR ACCEPTANCE OF THIS LICENSE AGREEMENT AND ESTABLISHES A BINDING AGREEMENT BETWEEN THE PERSON ACQUIRING THE SOFTWARE (THE “USER”) AND BROOKTROUT TECHNOLOGY (“BROOKTROUT”). IF YOU DO NOT ACCEPT THE LICENSE AGREEMENT, YOU SHOULD PROMPTLY RETURN THE SOFTWARE UNUSED AND YOUR MONEY WILL BE REFUNDED.

Brooktrout Technology Multi-Use License Agreement

Proprietary Rights

The Software is subject to the protection of the copyright laws of the U.S. and foreign jurisdictions, which prohibit unauthorized copying and distribution of copyrighted works. Certain uses of the Software are covered by one or more of the patents listed on the media and associated packaging. The Software incorporates proprietary and confidential algorithms and techniques that are subject to legal protection as trade secrets. Brooktrout is the sole owner of all proprietary rights in the Software, except for certain portions that are proprietary to third party licensors of Brooktrout. The User is granted only those rights expressly conferred by this License Agreement.

License

Brooktrout licenses the User to use the Software subject to and in accordance with the following provisions. The software is distributed with voice and/or fax processing boards and/or other computer hardware manufactured and sold by Brooktrout ("Brooktrout Hardware"), and is licensed solely for use in connection with the Brooktrout Hardware. The Software, and modified versions thereof, may be operated only on the central processing unit of any computer served by one or more items of Brooktrout Hardware and may, where appropriate in connection with such use, be downloaded into memory located on Brooktrout Hardware, and may be modified (if modification is otherwise permitted pursuant to the following provisions), reproduced and distributed only for purposes of such use. Any other use, modification, reproduction or distribution is expressly prohibited.

Licensing provisions applicable to particular Software products are as follows:

1. API, Application, and Driver software distributed in the form of object code:
    a. Users may incorporate the Software into their own work providing functional and value enhancements and may duplicate and distribute the resulting work as they choose provided that the resulting work is designated solely for use in connection with Brooktrout Hardware.
    b. Users may not modify the Software nor decompile, reverse engineer, disassemble, or otherwise reduce the Software to a human perceivable form.
    c. When Users incorporate the Software into their products, Brooktrout's copyright notice must be included in the new work.
    d. With respect to algorithms (such as, but not limited to, voice encoding methods, facsimile processing, and tone processing) included in the Software that are used for operations on the Digital Signal Processors (DSPs) of the Brooktrout Hardware, (i) Users may only use or incorporate such algorithms as components of the Software in the form originally supplied by Brooktrout, and may not extract any such algorithms from the Software or use them for any other purpose or permit any end user customer of the Users to have direct access to such algorithms for any purpose; and (ii) Users may only use or incorporate into their products those algorithms that have been licensed under a purchase agreement between the Users and Brooktrout in connection with the specific items of Brooktrout Hardware on which the Software containing such algorithms is to be distributed, and only to the extent of the quantity of Brooktrout Hardware having a total number of ports equal to the number of ports for which the licenses to such Software and the included algorithms have been paid.
2. API, Application, and Driver software distributed in the form of source code:
    a. Users may modify the Software and must incorporate it into their own work to provide functional and value enhancements. Users may duplicate and distribute the resulting work in object code form only, provided that the resulting work is designated solely for use in connection with Brooktrout Hardware. Users may not distribute the Software in source form.
    b. The Software is confidential and proprietary to Brooktrout and Users must protect it in a manner similar to the protection they affords their own confidential and proprietary information.
    c. When Users incorporate the Software into their products, Brooktrout's copyright notice must be included in the new work.

d. With respect to algorithms (such as, but not limited to, voice encoding methods, facsimile processing, and tone processing) included in the Software that are used for operations on the Digital Signal Processors (DSPs) of the Brooktrout Hardware, (i) Users may only use or incorporate such algorithms as components of the Software in the form originally supplied by Brooktrout, and may not extract any such algorithms from the Software or use them for any other purpose or permit any end user customer of the Users to have direct access to such algorithms for any purpose; and (ii) Users may only use or incorporate into their products those algorithms that have been licensed under a purchase agreement between the Users and Brooktrout in connection with the specific items of Brooktrout Hardware on which the Software containing such algorithms is to be distributed, and only to the extent of the quantity of Brooktrout Hardware having a total number of ports equal to the number of ports for which the licenses to such Software and the included algorithms have been paid.

The reproduction, distribution, and modification rights provided above apply only to Brooktrout Software packaged herewith that bears the label “Multi-Use License Agreement applies.” All other Software distributed by Brooktrout is subject to additional restrictions on reproduction, distribution, and modification.

Distribution

Any distribution of the Software (including modified versions) that is authorized hereby shall be made (a) in object form only; (b) only to purchasers of units of Brooktrout Hardware, or of products including Brooktrout Hardware, for which appropriate payment (including payment for the Software that has been distributed) has been made in accordance with the purchase agreement between the Users and Brooktrout and (c) only pursuant to license agreements containing, at a minimum, the following provisions: (i) express acknowledgement of Brooktrout’s and its licensors' proprietary rights in the Software, (ii) a license to use the Software only as installed on the units of Brooktrout Hardware purchased by the licensee, (iii) the express prohibition of any reproduction, modification, or distribution of the Software, (iv) a prohibition on reverse engineering of the Software equivalent to that set forth above, and (v) provisions regarding Termination, Disclaimer of Warranties, and Limitation of Liability, expressed for the benefit of Brooktrout, substantially in the form set forth below. Except as expressly permitted hereby, Users may not distribute the Software, or any copy, by transfer, lease, loan or any other means.

Termination

A User's license to use the Software may be terminated by Brooktrout in the event of any failure to comply with the above restrictions or any other terms of this License Agreement. In the event of termination of the license, the User must destroy or return to Brooktrout all copies of the Software in his possession.

Limited Warranty

Brooktrout warrants for a period of 90 days following delivery that the media on which the Software is recorded is free from defects in materials and workmanship. Brooktrout does not warrant that operation of the Software will be uninterrupted or error-free, or that it will satisfy the User's requirements. BROOKTROUT DISCLAIMS ALL OTHER WARRANTIES EXPRESSED OR IMPLIED, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Limitation of Liability

Brooktrout's entire liability and the User's exclusive remedy in connection with the Software will be the replacement of any media not meeting the above limited warranty upon return of the media to Brooktrout. In no event will Brooktrout be liable for damages, including any lost profits or other incidental or consequential damages, arising out of or related to the Software and its use, even if Brooktrout has been advised of the possibility of such damages.

Table of Contents

List of Figures

Figure **Page**

List of Tables

Table **Page**

List of Examples

Example **Page**

Preface

This manual describes how to write applications using the RealCT Direct API. It includes information about the architecture of the API, and how to use the MVIP-90 and H.100 computer telephony buses.

This chapter contains the following sections:

- About this Manual
- Getting Help

About this Manual

This guide contains the following chapters:

Chapter 1	Describes how to write applications using the application architecture.
Chapter 2	This chapter describes how to send and receive digit. It provides information about rotary, DTMF, MF, and R2 signaling.
Chapter 3	This chapter describes T1 networking.
Chapter 4	This chapter describes E1 networking.
Chapter 5	Describes how to use the MVIP-90 bus to switch data between telephony resources.
Chapter 6	Describes how to use the MVIP-95 bus to switch data between telephony resources.
Appendix A	This appendix describes the signaling bits transmitted in the most common T1 line protocols.
Appendix B	This appendix describes the signaling bits transmitted in the most common E1 line protocols.

Typographical Conventions

This manual uses the typographical conventions shown in the following table:

Table 1. Typographical Conventions

Convention	Type of Information	Examples
Italic type	A book or a manual A file, path, or program A data structure	*Voice Utilities User Guide* C:\RHT\ *VPconfigCallerID_s*
`Monospaced font`	Code A command	`hVP = CreateFile` `HANDLE hVP`
SMALL CAPS	A code constant	BRKT_DEVICE_VP
Single quotes	A digit A field	'7' 'Timeout'

Related Manuals

Refer to the following documents for additional information as you develop computer telephony applications:

- *Introduction to Computer Telephony*

  Provides an introduction to computer telephony.
- The hardware installation guide for your hardware type

  Describes how to install hardware into your system.
- *RealCT Direct API Installation and Configuration Guide*

  Explains how to install and configure your software and hardware.
- *RealCT Direct API Reference Manual* (two volumes)

  Describes all functions and data structures in the RealCT Direct API.
- *Voice Utilities User Guide for Windows Environments*

  Contains information about AccuCall Plus, AccuCall Wizard, CTMedic, ShowJump, and Prompt Studio.

Documentation Feedback

Brooktrout is committed to continuously improving the technical accuracy and usability of our product documentation. All suggestions, comments, or corrections are welcome. Send your feedback to *emgtechpubs@brooktrout.com*. Include the following information in your correspondence:

- Document number, located on the title page
- Release date, located on the title page
- Your telephone number if you would like us to contact you personally

Your comments help us provide the most accurate documentation possible.

Getting Help

If you encounter problems, contact Brooktrout technical support. You must give technical support your customer number. If you do not know your customer number, you can get it from your Brooktrout sales representative.

U.S.A.

Brooktrout Technology. VTD
151 Albright Way
Los Gatos, California 95032

Phone: +1 408 874-4040
8:00 A.M. to 5:00 P.M. Pacific time, Monday through Friday
Fax: +1 408 370-1171
Email: support@brooktrout.com
Website: www.brooktrout.com

United Kingdom

Brooktrout Technology
Centennial Court
Easthampstead Road
Bracknell, Berkshire RG12 1YQ
United Kingdom

Phone: +44 1344 380 280
Fax: +44 1344 380 288

1 Architecture

This chapter describes how the API interacts with the operating systems that this API supports, discusses how the systems are designed, and explains how the components of the system are connected to and operate with each other.

This chapter includes the following topics:

- How RealCT Interfaces With Your System
- Reviewing the Key System Components
- Application Architecture

How RealCT Interfaces With Your System

Interfacing With Windows Platforms

The Win32 Application Programming Interface (API) provides a set of functions and defines how those functions behave. Microsoft provides two platforms that support the Win32 API: Windows NT and Windows 2000. Since each of these platforms support a common set of Win32 API functions, you can write one application for all platforms. However, the three platforms implement functions differently and to different degrees.

Windows NT

Windows NT supports the full functionality of the Win32 API. This full support means that Windows NT applications are robust and fast. Windows NT can also run on machines with different CPUs such as MIPS R4000, a DEC alpha, or Motorola's PowerPC.

Windows 2000

To be supplied.

Header File

To interface to the Brooktrout Direct device manager library, your application must include the following header file:

```
"brktddm.h"
```

This header file also includes other header files that define function tags, data structures, parameter values, and error values for the device drivers.

Library

You must compile and link your application program with the RealCT Direct Device Manager (DDM) library. Brooktrout recommends using the Microsoft C/C++ compiler, versions 6.0 and later, for the RealCT Direct API. The library file is:

```
brktddm.lib
```

Note: Ensure that you have installed the **`brktddm.dll`** file during the installation process.

Reviewing the Key System Components

A PC-based computer telephony system consists of the following basic components as shown in Figure 1:

- One or more computer telephony boards
- Firmware on the board
- Device drivers
- RealCT Direct API (Application Programming Interface) files
- Application source code

During installation, download the firmware to the digital signal processor (DSP) on the board and load the device drivers. Your application makes a call to the device driver through the Brooktrout DirectDM library. In turn, the device driver communicates with the firmware on the DSP.

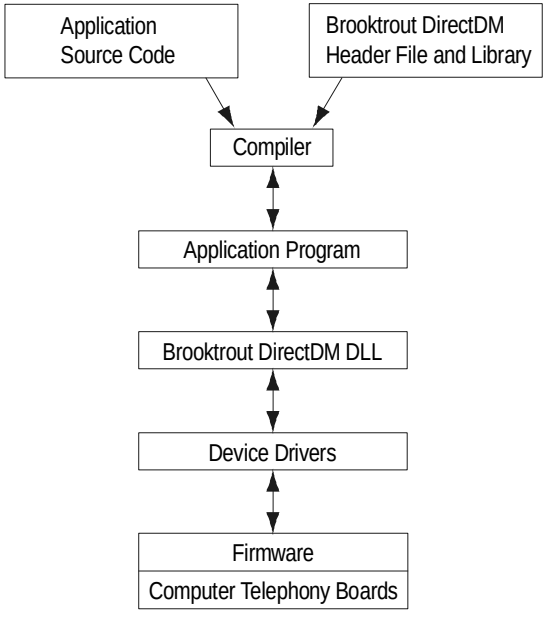

Figure 1. Basic Components of a Computer Telephony System

Computer Telephony Boards

Voice processing (VP) boards, such as the Prelude, RDSP, or Vantage products, provide the voice processing and telephony hardware for your computer telephony system.

The RTNI series boards provide either analog or digital line capabilities that combine with a voice processing board to expand and enhance the versatility of the voice processing system. The PRI-ISALC boards provide single or dual span E1 and T1 digital line capabilities.

The RealBLOCs series board features modules that provide analog trunk, analog station, or passive tap lines on a single board.

The DST-16 provides digital station passive tap capability for call recording applications when combined with a Vantage VRS board. Different versions of the board provide support for the following PBXs:

- Nortel M1
- Nortel Norstar
- NEC
- Aspect
- Lucent/Siemens 2 wire
- Lucent 4 wire
- Panasonic

Digital Signal Processor

The Digital Signal Processor (DSP) is a microprocessor dedicated to manipulating digital signals. Each DSP supports four voice processing (VP) resources, also referred to as channels. A board such as the RDSP/432, which provides two or four VP resources, has only one DSP. A board such as the VRS-32, which provides 32 VP resources, has eight DSPs.

Firmware

The firmware is a set of algorithms that run on the DSP. The firmware provides voice processing features such as tone detection and generation, file compression and expansion, pulse detection, and playback speed and volume control. The firmware algorithms operate simultaneously but independently of each other, allowing boards to perform multiple functions.

You download firmware to the DSP when you load the drivers.

Devices

The API identifies devices and boards by a device name as follows.

Table 2. Device Type Names and Boards

Device	Device Type Name	Telephony Boards
Voice processing	BRKT_DEVICE_RDSP_BOARD BRKT_DEVICE_RDSP BRKT_DEVICE_VP	Prelude Series RDSP 400 Series RDSP 9400 Series RDSP 24000 Series Vantage™ PCI Series Vantage Volare Vantage VPS Series Vantage VRS Series
Loop start line	BRKT_DEVICE_LS_LINE	Prelude Series RDSP 400 Series RDSP 9400 Series Vantage PCI Series Vantage Volare Vantage VPS Series
Analog station interface and analog loop start trunk interface	BRKT_DEVICE_ATSI_BOARD BRKT_DEVICE_ASI_LINE BRKT_DEVICE_ATI_LINE	RTNI-ATI/ASI Series RealBLOCs ASI and ATI PCI Series
Digital E1	BRKT_DEVICE_E1_BOARD BRKT_DEVICE_E1_LINE	PRI-ISALC-1E or E or E1
Digital T1	BRKT_DEVICE_T1_BOARD BRKT_DEVICE_T1_LINE	PRI-ISALC-1T or -2T or RTNI-2T1

Voice processing boards provide voice resources on the digital signal processors (DSP) and loop start (LS) line resources on the line interfaces (except the RDSP 24000 and Vantage VRS series boards that provide only voice resources). Further, each DSP is logically divided into four VP resources. Figure 2 shows this division of the hardware and logical resources on a typical board

(RDSP/432). More information about the capabilities of each board can be obtained in the *Introduction to Computer Telephony* manual.

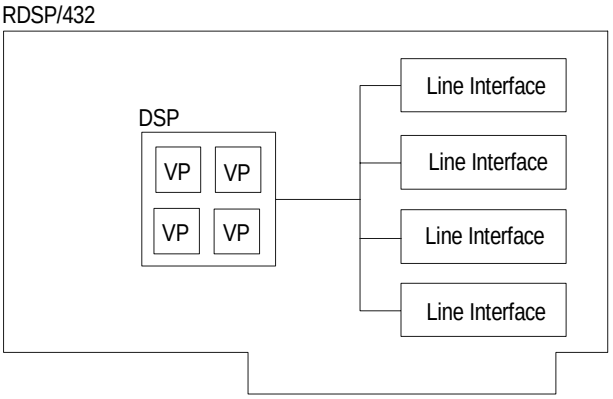

Figure 2. Hardware and Logical Resources on an RDSP/432 Board

The API models these hardware and logical resources as *devices*. Table 3 shows how the hardware and logical resources of the RDSP/432 board map to devices.

Table 3. Hardware/Logical Resources Mapped to Devices

Hardware and Logical Resource	Device	Device Name
VP board DSP	Board device DSP device	BRKT_DEVICE_RDSP_BOARD BRKT_DEVICE_RDSP
4 VP resources	VP devices	BRKT_DEVICE_VP
4 LS lines	LS line devices	BRKT_DEVICE_LS_LINE

Application Programming Interface

The Application Programing Interface (API) is the interface between an application and the device drivers. The interface is a set of commands the application issues to the device driver. The device driver then sends the commands to the firmware on the board. The RealCT Direct API is defined by a set of C header files consisting of the following:

- Function Tags
- Data structures
- Parameters
- Errors

You can find detailed information about the RealCT Direct API in the *RealCT Direct API Reference Manual.*

Function Tags

Function tags are the tags used with the *BrktDeviceIoControl ()* API call. Each of the device drivers has exclusive and non-exclusive function tags. Exclusive functions prevent other exclusive functions from running simultaneously on a specific device. This means that you can call only one exclusive function at a time for any specific device.

Function tags used to set parameters or retrieve status from the driver are non-exclusive. You can call non-exclusive functions while an exclusive function is running.

Data Structures

The data structures contain the parameters and data used by or returned by the drivers. When you call a driver function that passes parameter values like RHT_SET_GLOB or RHT_SET_PARAM in its data structure, these values override standard parameter values set when the device driver is loaded.

Parameters

The API includes both global and device-specific parameters that determine the behavior of the driver functions. Each time you load the device drivers, these parameters are automatically set to the values specified through the Configuration Wizard.

You can set any parameter value for your application by calling the RHT_SET_GLOB or RHT_SET_PARAM function tags. The application uses the new values unless an overriding value for RHT_SET_GLOB or RHT_SET_PARAM is passed in a data structure. Data structure parameter values override set parameter values for the duration of the function call. The set parameter values remain in effect for as long as the device driver is loaded or until they are reset through another call.

Errors

The device drivers set an error code when errors occur. Use *BrktGetLastError ()* to get the error value for the last system call or the driver function that returned a value of FALSE.

The application uses the standard operating system error return codes to report the status of the API call to the application.

API Calls

RealCT Direct uses the API calls listed in Table 4 to communicate with the devices.

Table 4. API Calls

System Calls	Description
BrktOpenDevice ()	Opens the device file.
BrktCloseDevice ()	Closes the device file.
BrktDeviceIoControl ()	Sends the function tag to device driver to perform specified operation.
BrktGetLastError ()	Returns the last operating system error code for calling thread.

BrktDeviceIoControl ()

The *BrktDeviceIoControl ()* API call sends a function tag to the device driver, causing the device to perform the specified operation. If the function succeeds, then the return value is TRUE. If the function does not succeed, it returns FALSE. If the function returns FALSE, use the *BrktGetLastError ( )* function to find out what error occurred.

Application Architecture

Processes and Threads

An instance of an executing application is a process. For example, a word processing program and a spreadsheet that are executing simultaneously are both processes. You can have many processes running at once. Each process consists of data, code, and one or more threads. The processes cannot share data directly.

Each process can have associated threads that execute functions of the process. Each process can have more than one thread, each of which has CPU registers and a stack. The threads share data as shown in Figure 3.

For example, within the word processing process, you can have threads for printing, saving, and receiving keyboard data. These threads can execute simultaneously. The first thread, called the primary thread, is created when the application starts and starts executing from the program's entry point. This thread can generate new threads, which can generate threads of their own.

A process must have at least one thread. When the last thread terminates, the process also terminates, but terminating the primary thread does not terminate the process as long as there is one thread executing. When the process terminates, all threads associated with that process also terminate.

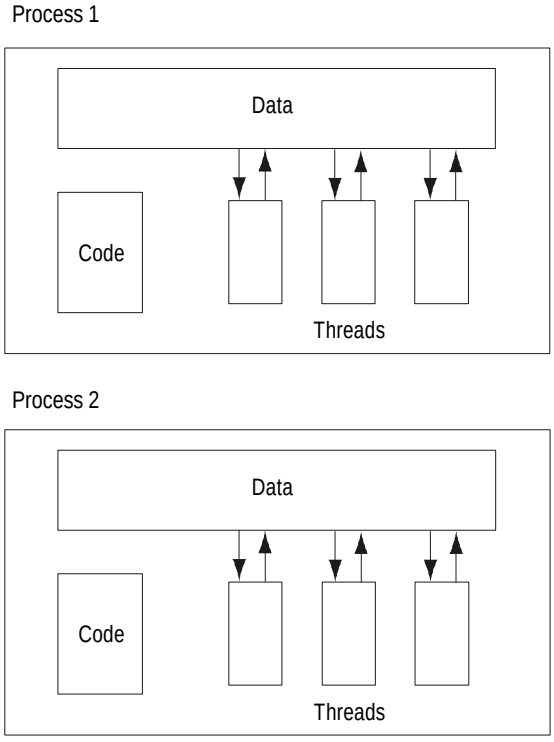

Figure 3. Two Processes With Threads

Application Design

Although you can call the *BrktDeviceIoControl ( )* function in either synchronous or asynchronous mode, we only recommend using synchronous mode. This means that when the application calls a function, it cannot proceed until that function completes. If you use one thread and one process, a call on line one prevents the application from processing calls on other lines.

There are two ways to design an application with synchronous drivers: with or without multi-threading.

Non-multithreaded

Non-multithreaded applications run a separate instance of a process for each channel. For example, the four-port RDSP/432 would require four instances of the same telephony application. With this model, any functions running in the first instance of the application do not block another instance from processing a call on a different line.

Non-multithreaded applications are quick to write and are quite stable, since if one instance of the process terminates unexpectedly, all other processes are not affected. However, running several instances of a single process uses system resources inefficiently. Also, since different processes don't easily share data with each other, applications that require knowledge of the status of two or more channels, such as switching functions, are more difficult to implement.

Multithreaded

Multithreaded applications use system resources more efficiently. In multithreaded applications, a single process runs multiple threads. Each thread executes a separate function. When the application begins, it immediately starts one thread for each channel. These threads constantly monitor the line. Then, if a particular thread needs to execute simultaneous functions, it can start additional threads.

For example, when you call a bank to retrieve information, a single thread requests your user information, then provides customer options. If you request information from a database, that single thread creates a second thread to search the database while the original thread plays pre-recorded music. Since separate threads execute the two functions, they do not block each other. An application that is not multithreaded cannot run the two functions simultaneously.

Multithreading provides flexibility in the application and allows threads to share data more easily. However, if one thread causes an exception error, the entire application fails. Synchronization can also become an issue.

2 Digit Handling

This chapter describes how to send and receive digits. It provides information about rotary, DTMF, MF, and R2 signaling.

All boards that provide VP resources can send and receive digits.

This chapter includes the following sections:

- Using Digits
- Defining Digit Types
- Sending and Receiving Digits
- Troubleshooting

Using Digits

A person using a telephone or a telephony application begins a call by seizing the line, then sending digits to the receiving end. When you place a call on a standard analog phone, these digits are the ones you dial on your telephone key pad. They tell the central office (CO) where to direct the call. The CO then communicates with the other COs in the public network to route the call. Once the call is established, the receiving end processes the call. During a call, you can navigate through information or control functions of a voice mail application using the same digits.

There are four common ways to send digits: Rotary, Dual Tone Multi-Frequency (DTMF), Multi-Frequency (MF), and R2 compelled signaling.

- Rotary digits are the signals sent by rotary phones or by your touch tone phone when you set it to send rotary signals.
- DTMF tones are on your telephone key pad.
- MF and R2 signals are used in communication between two COs or between COs and digital telephony boards.

Brooktrout boards with voice processing capabilities can handle the four types of signals regardless of the interface.

Defining Digit Types

The four methods of sending digits all convey the same information to the CO about which digit the transmitting end dialed. Some methods, such as R2 signaling, convey additional information about the call such as call priority or line status.

When you dial a number on a standard analog phone, the CO receives the digits and routes the call to the appropriate destination. If necessary, the CO communicates with COs in route to the far end using the appropriate signaling method for that line such as MF or R2 digits. For example, if you call a telephony application that uses a T1 line, you dial DTMF digits on your touchpad. The CO then establishes the call with the far end using MF digits. Once the audio path is established, you navigate the application using DTMF digits.

Defining Rotary Digits

Rotary digits are the ones you dial on a rotary phone. As the dial turns, the loop current is interrupted once for each number. For example, when you dial '7', the telephone briefly opens and closes the hook relay seven times, which results in pulses of loop current. You hear two ticking sounds for each pulse, one when the loop current is interrupted and one when it is reestablished, for a total of 14 ticks when you dial the number '7'.

Although you can place calls using rotary digits, not all telephony applications recognize rotary digits so you cannot always use them to navigate through the system.

Defining DTMF Signals

DTMF digits are the ones on your telephone keypad. Figure 4 shows the combination of two out of eight possible frequencies that make up each digit. All digits in a given row share the same low frequency. All digits in a given column share the same high frequency.

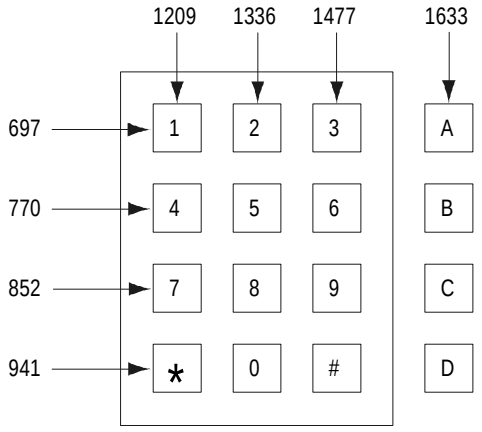

Figure 4. Frequencies in DTMF Digits

For example, if you press ‘7’, your phone sends a dual-frequency tone made up of 852 Hz and 1209 Hz to the CO. When you dial out, the CO uses the digit as part of an address. During a call, applications recognize DTMF tones as a way to navigate through information. All Brooktrout boards can recognize and send DTMF digits.

DTMF digits include the standard twelve keypad digits, along with additional ‘A’, ‘B’, ‘C’, and ‘D’ signals. Phone systems use these additional signals to assign priority to a call.

Defining MF Signals

MF digits use combinations of 6 frequencies to make a total of 15 dual-frequency tones, as Table 5 shows. MF signals are also called R1-MF.

Table 5. MF Tones

Digits	Frequencies					
	700	900	1100	1300	1500	1700
1	x	x				
2	x		x			
3		x	x			
4	x			x		
5		x		x		
6			x	x		
7	x				x	
8		x			x	
9			x		x	
0				x	x	
spare	x					x
spare		x				x
KP			x			x
spare				x		x
ST					x	x

COs in the United States, Canada, and Japan generally use MF digits to establish calls. Telephony systems that use T1 lines also use MF digits to communicate with the CO. Once the call is established, callers use DTMF digits to navigate telephony applications.

MF signaling uses five signals in addition to digits 0 through 9. The key pulse (KP) signal comes at the beginning of any digit string and prepares the receiving end to accept the digits. The start signal (ST) signal comes at the end of any digit string. The remaining three signals (spare) are reserved for the national identity of the receiving end.

Defining R2 Signals

COs outside the United States, Canada, and Japan generally use R2 digits to establish calls. Digital telephony systems in these countries also use R2 digits to communicate with the CO. R2 signals, which are also called R2-MF, involve a compelled handshake between the transmitting and receiving ends to confirm that each end receives the signal. Once the call is established, callers use DTMF digits to navigate telephony applications.

Although R2 signals are often used over E1 lines, there is no relationship between the R2-CCITT line protocols used by E1 lines and the R2 signaling.

Timing the Signals

In R2 inter-register signaling, each signal from the transmitting end requires an acknowledgment from the receiving end. The handshaking process ensures that both ends receive the signal properly.

The end that originates the call has outgoing registers and sends forward inter-register signals. The end that receives the call has incoming registers and sends backward inter-register signals. A compelled signal between outgoing and incoming registers uses the following steps:

1. The outgoing R2 register starts transmitting a forward signal.
2. The incoming R2 register sends a backward acknowledgment signal as soon as it receives the forward signal.
3. The outgoing R2 register stops sending the forward signal when it receives the acknowledgment.
4. The incoming R2 register stops sending the acknowledgment when it stops receiving the forward signal.
5. The outgoing register transmits the next R2 signal when it stops receiving the previous backward acknowledgment signal.

Figure 5 shows the sequence of events in R2 compelled signaling.

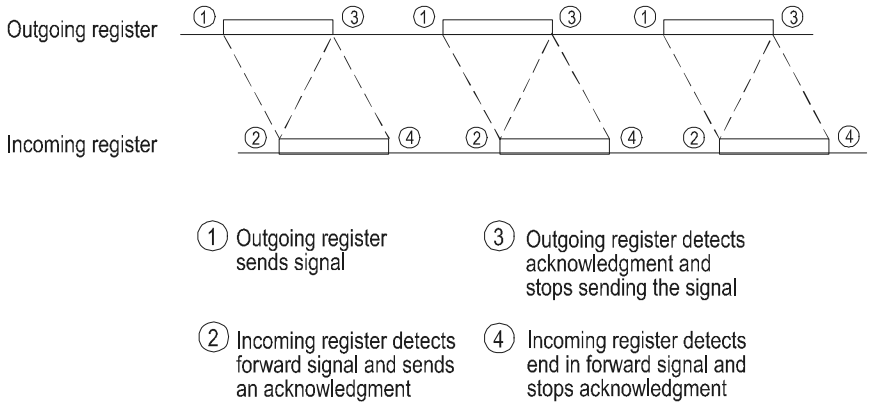

Figure 5. The R2 Compelled Signaling Sequence

Each end waits for an acknowledgment or end of signal until it times out. This handshaking makes compelled signaling more robust than other non-compelled signaling methods.

The speed of compelled signaling depends on the quality of the link. If the lines are good, then detection time and overall performance is fast, but bad lines delay detection and slow down the signaling. In non-compelled signaling, you would have to set the digit duration high enough to account for bad lines, which makes signaling consistently slow even if the lines are good.

Assigning Frequencies

The forward and backward signals use a different set of six frequencies for a total of 15 possible forward and backward tones.

- Forward frequencies: 1380, 1500, 1620, 1740, 1860 and 1980 Hz
- Backward frequencies: 1140, 1020, 900, 780, 660 and 540 Hz

Networks can also operate using only five forward frequencies for a total of ten signals, and four or five backward frequencies for a total of six or ten signals. Mexico and China, for example, use only four backward frequencies.

Table 6 shows how the frequencies combine to form signals 1 through 15 for forward and backward signals. Each frequency is assigned a value for index (x) and a weight (y). As Table 6 shows, adding the index value and the weight value gives a number from 1 to 15. For example, to form the digit one, add the frequency for x=0 and y=1, or 1380 Hz and 1500 Hz for a forward signal.

Table 6. R2 Tones

Combination		Frequencies						
x+y=	No.	Forward	1380	1500	1620	1740	1860	1980
		Backward	1140	1020	900	780	660	540
		Index (x)	x=0	x=1	x=2	x=3	x=4	x=5
		Weight (y)	y=0	y=1	y=2	y=4	y=7	y=11
0+1=	1		x	y				
0+2=	2		x		y			
1+2=	3			x	y			
0+4=	4		x			y		
1+4=	5			x		y		
2+4=	6				x	y		
0+7=	7		x				y	
1+7=	8			x			y	
2+7=	9				x		y	
3+7=	10					x	y	
0+11=	11		x					y
1+11=	12			x				y
2+11=	13				x			y
3+11=	14					x		y
4+11=	15						x	y

Meaning of R2 Frequencies

The forward and backward signals each have two different meanings, depending on when the signal is sent. The forward signals are divided into Group I and Group II. The backward signals are divided into Group A and Group B.

The first forward signal always has the Group I meaning, which the incoming register acknowledges with a Group A response. To switch over to using Group II and Group B meanings, the incoming register sends a signal that prepares the outgoing register to both send Group II signals and receive Group B signals. When the signaling switches groups, the frequencies used remain the same but their meaning changes. In systems with only four or five backward frequencies, forward and backward signals are usually divided into three groups. Increasing the number of groups makes up for the smaller number of available signals.

Since each tone has two meanings, R2 signaling can convey more information than other signaling methods without using additional frequencies.

Some signals are reserved to be defined locally, so check with the local carrier for information about what the signals mean in that area. Because of regional differences, you might need to customize applications for each country.

Group I Signals

Table 7 shows the meaning of the Group I signals as they are defined in the ITU-T standards. These signals convey information about the country or the language of the originating call or the digits in a number. The meaning of Group I signals depends on whether they are sent at the beginning of a call or in response to a Group A request for information.

Table 7. Group I Signals

Signal	Meaning 1	Meaning 2
1	Language: French	Digit 1
2	Language: English	Digit 2
3	Language: German	Digit 3
4	Language: Russia	Digit 4
5	Language: Spanish	Digit 5
6	Spare (language)	Digit 6
7	Spare (Language)	Digit 7
8	Spare (Language)	Digit 8
9	Spare (Discriminating)	Digit 9
10	Discriminating digit	Digit 0
11	Country code: outgoing half-echo suppressor required	Access to incoming operator
12	Country code: no echo-suppressor required	i) Access to delay operator ii) Request not accepted
13	Test call indicator	i) Access to test equipment ii) Satellite link not included
14	Country code: outgoing half-echo suppressor inserted	i) Incoming half-echo suppressor required ii) Satellite link included
15	Signal not used	End of identification

Group II Signals

The outgoing register switches to Group II signals after the incoming register sends signal A-3 or A-5. These signals tell the receiving end the call category: subscriber, operator, or data. Table 8 shows the meanings of the Group II signals.

Table 8. Group II Signals

Signal	Meaning	Comments
1	Subscriber without priority	Used nationally
2	Subscriber with priority	
3	Maintenance equipment	
4	Spare	
5	Operator	
6	Data transmission	
7	Subscriber, or operator without forward transfer facility	Used internationally
8	Data transmission	
9	Subscriber with priority	
10	Operator with forward transfer facility	
11	Spare for national use	
12		
13		
14		
15		

Group A Signals

The incoming register sends Group A signals in acknowledgment to the Group I signals sent by the outgoing register. These signals request additional information from the transmitting end. Signals A-3 and A-5 prepare the outgoing register to both receive Group B signals and send Group II signals. Systems that use 6 or 10 backward signals have significantly different meaning for the signals. Table 9 shows the meanings of the Group A signals in a system that uses 15 signals.

Table 9. Group A Signals

Signal	Meaning
1	Send next digit (n+1)
2	Send last but one digit (n-1)
3	Change to receive Group B signals
4	Congestion in the national network
5	Send calling party's category
6	Address complete, charge, set up speech conditions
7	Send last but two digit (n-2)
8	Send last but three digit (n-3)
9	Spare for national use
10	Spare for national use
11	Send country code indicator
12	Send language or discriminating digit
13	Send nature of circuit
14	Request information about whether an incoming half-echo suppressor is required
15	Congestion in an international exchange or at its output

Group B Signals

The incoming register sends Group B signals after sending A-3 or A-5. The Group B signals provide information about the subscriber's line. Table 10 shows the meanings of the Group B signals.

Table 10. Group B Signals

Signal	Meaning
1	Spare for national use
2	Send special information tone
3	Congestion encountered after changing from Group A to Group B signals
4	Unallocated number
5	Subscriber's line free, charge
6	Subscriber's line free, no charge
7	Subscriber's line out of order
8	Spare for national use
9	
10	
11	
12	
13	
14	
15	

Sending and Receiving Digits

Handling Rotary Digits

Boards with VP resources can receive rotary digits but cannot dial rotary digits.

Each phone is connected to a line with unique electrical characteristics, and phones send digits with slightly different timing. For the application to properly receive rotary digits from different phones, it must train to recognize the way that phone sends rotary pulses. The best way to recognize the digits is to train at the beginning of a call by asking the caller to press '0' if they have a rotary phone. The application then uses the '0' digit to learn the timing of that phone. The application can then recognize any rotary digits from that phone.

Use the following procedure to receive rotary digits:

1. Specify what type of digits the application receives using RHT_SET_DIGIT_MODE.

   Set the mode to EN_ROTARY to receive rotary digits. You can specify that your application also receive DTMF or MF digits.

2. Train rotary digits using RHT_TWAIT_DIGIT or RHT_PRE_REC.

   These functions both use the data structure *VPstartStop_s*, which has a field for rotary training. Set that field to train on digit '7', '8', '9', or '0'. For most operations, '0' provides the best training. In the application, ask customers to dial the digit you specify in *VPstartStop_s*.

3. Check the status of rotary training using RHT_GET_ROTARY_INFO.

4. Receive rotary digits using RHT_TWAIT_DIGIT, RHT_READ_DIGIT, RHT_PLAY, or RHT_REC.

   Once you have trained the application to recognize pulse characteristics from that phone, it can recognize any rotary digit.

Handling DTMF and MF Digits

To send and receive DTMF and MF tones, use the following procedure:

1. Specify what type of digits the application receives using RHT_SET_DIGIT_MODE.

   Set the mode to EN_DTMF for DTMF digits or EN_MF for MF digits.

   You can enable rotary detection with DTMF or MF detection, but the application cannot detect both MF and DTMF digits. The two signaling methods use similar frequencies, which could lead to inaccurate signal detection.

2. Receive tones using RHT_TWAIT_DIGIT or RHT_READ_DIGIT.

   These functions use the RHT_SET_DIGIT_MODE to determine what type of digit to receive. RHT_TWAIT_DIGIT and RHT_READ_DIGIT clear the digit from the digit buffer.

3. Send digits using RHT_DIAL.

   If you specify a VP device, RHT_DIAL sends DTMF digits by default. To send MF digits, use an 'M' in the string passed to RHT_DIAL. To specify DTMF digits, use a 'T' in the string. For example, the string "123M456T7" specifies DTMF digits 1, 2, 3, then MF digits 4, 5, 6, then DTMF digit 7.

Handling R2 Digits

To send and receive R2 digits, use a similar procedure to handling DTMF and MF digits, except use RHT_DIAL_R2 to send R2 tones. RHT_DIAL_R2 handles all aspects of the compelled protocol. When it returns, the application can proceed to send or receive the next signal.

Send R2 digits in either compelled or pulse mode. In compelled mode, RHT_DIAL_R2 sends the digit until it detects the appropriate acknowledgment from the other end. In pulse mode, the function sends the digit for an amount of time specified by *VPdialR2_s.Pulse*, regardless of signals received from the far end.

Before sending or receiving digits, use RHT_SET_DIGIT_MODE to specify whether the application detects forward or backward signals. RHT_DIAL_R2 also uses this setting to determine which digits to send. If RHT_SET_DIGIT_MODE specifies to detect backward signals, then RHT_DIAL_R2 dials forward signals and vice versa.

The following steps show how to handle R2 signaling if the application receives a call:

1. Set RHT_SET_DIGIT_MODE to EN_R2_FORWARD.

   The application receives forward signals and sends backward signals.
2. Specify the maximum time to wait for the forward digit to terminate using RHT_SET_GLOB.
3. Detect forward digits using RHT_TWAIT_DIGIT with a digit count of 1 and a timeout of 15 seconds.

   RHT_TWAIT_DIGIT returns when it detects a forward digit. It clears the digit from the digit buffer.
4. Transmit the appropriate backward signal using RHT_DIAL_R2.

   If RHT_DIAL_R2 returns successfully and more signals are expected, return to step 3.

The following steps show how to handle R2 signaling if the application transmits a call:

1. Set RHT_SET_DIGIT_MODE to EN_R2_BACKWARD.

   The application receives backward signals and sends forward signals.
2. Specify the maximum time to wait for a backward digit using RHT_SET_GLOB.
3. Transmit the forward signal using RHT_DIAL_R2.

   RHT_DIAL_R2 terminates when it detects the backward signal.
4. Read the digit using RHT_TWAIT_DIGIT or RHT_READ_DIGIT with a digit count of 1.

   Since RHT_DIAL_R2 has already detected a digit, RHT_TWAIT_DIGIT should return immediately and successfully. RHT_TWAIT_DIGIT clears the digit from the digit buffer.
5. If the application needs to send more digits, return to step 3.

Example 1 shows how to send digits using RHT_DIAL_R2.

Example 1. RHT_DIAL_R2 Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE VpHandle;           /* VP device handle */
   BOOLEAN IoctlResult;            /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;      /* Bytes returned from IOCTL call*/
   USHORT DigitMode;
   struct VPdialR2_s Dial;

   /* Open VP device */
   VpHandle = BrktOpenDevice (BRKT_DEVICE_VP, 0);

   /* Set digit mode to detect forward R2 digits */
   DigitMode = EN_R2_FORWARD;

   IoctlResult = BrktDeviceIoControl (
         VpHandle,
         RHT_SET_DIGIT_MODE,
         &DigitMode,               /* Buffer to driver */
         sizeof(DigitMode),
         NULL,
         0,
         &BytesReturned,
         NULL);                    /* Wait until I/O is complete */
```

Example 1. RHT_DIAL_R2 Sample Code (Continued)

```
   /* Dial R2 backward digit. Synchronous call. */
   memset (&Dial, 0 sizeof (Dial));
   Dial.R2Digit = 2;              /* Dial R2 backward digit 2 */
   Dial.Pulse = 0;                /* Use compelled mode */

   IoctlResult = BrktDeviceIoControl (
         VpHandle,
         RHT_DIAL_R2,
         &Dial,                   /* Buffer to driver */
         sizeof(Dial),
         NULL,
         0,
         &BytesReturned,
         NULL);                   /* Wait until I/O is complete */

   if (!IoctlResult)
         printf ("RHT_DIAL_R2 failed: BrktGetLastError = %d\n",
               BrktGetLastError (VpHandle));

   BrktCloseDevice (VpHandle);
}
```

Flushing the Digit Buffer

Before detecting new digits from an incoming call, flush the digit buffer using RHT_FLUSH_DIGIT. In a well designed application there should never be digits remaining in the buffer after a call, since the digits are deleted after the application reads them. However, if the application does not run properly, then extra digits might remain in the buffer even after you terminate the application, remove any bugs, and run it again.

Flushing the digit buffer is particularly important in handling R2 digits. If digits remain in the digit buffer from a previous call, the application inappropriately responds to the digits although no digits have been sent on the line. The resulting protocol violation causes the carrier to drop the call.

The best time to flush digits is after going on-hook but before monitoring for a new call using RHT_WAIT_LINE_ON. If you flush digits after calling RHT_WAIT_LINE_ON, your thread could get pre-empted. If that happens, the function might not execute until after the application has started receiving digits from a new call. When the application eventually flushes the digit buffer, it clears digits that are already stored so the application doesn't receive the entire digit string. In R2 signaling, the application waits for a new digit rather than acknowledging any of the deleted received digits. However, the other end doesn't send a new digit until it receives an acknowledgment. The function eventually times out when neither end receives the expected signal.

The timing of when to flush digits is less important if the application is sending the call. However, it is best to flush digits before beginning the call. This way, if you ever reuse the code as the receiving end, the application will not flush digits at the wrong time.

Troubleshooting

The RHT_DIAL_R2 function returns successfully when it sends the R2 signal and detects a digit of the opposite type. When RHT_DIAL_R2 returns, you should check the termination condition in RHT_GET_STATUS. Call RHT_GET_STATUS for either a VP or line device. It returns the VP termination type in the structure *VPchanStatus_s*, or the line termination type in *RTNI_lineStatus_s*.

Some possible VP termination conditions when making an R2 call include:

T_RHT_N_DIGITS — Indicates that RHT_TWAIT_DIGIT detected an R2 digit.

T_RHT_TIMEOUT — Indicates that RHT_DIAL_R2 did not detect a backward digit within the time specified by *VPP_R2_TIMEOUT*, or that it did detect a digit but the digit did not terminate within the time specified by *VPP_R2_TIMEOUT* after the forward signal stopped.

This termination condition should only happen with RHT_DIAL_R2, and not with RHT_TWAIT_DIGIT. Since the application doesn't call RHT_TWAIT_DIGIT until RHT_DIAL_R2 has detected a digit, RHT_TWAIT_DIGIT should never experience timeouts.

T_RHT_LINEOFF — Indicates the function terminated due to a condition detected on the line. In this situation, call RHT_GET_STATUS for the line device to find the termination type for the line. Some common termination types include:

LOOP_OFF — Indicates that the remote end disconnected. In this case, the application should terminate the call.

UNEXPECTED_CALL_ANSWERED — Indicates the far end answered the call during the R2 signaling. This termination generally indicates that you are trying to send too many digits.

For a complete list of error codes returned by RHT_GET_STATUS, see the *RealCT Direct API Reference Manual*. For most line terminations types, you should end the call using RHT_DISCONNECT.

3 T1 Networking

This chapter describes T1 networking in your system environment.

The T1 and NetAccess board provides two T1 trunks for high-speed communications. The T1 board provides only T1 line resources. It must be in a system with a board that provides voice processing resources over a CT bus such as the Vantage series products.

This chapter includes the following sections:

- Understanding T1 Trunks
- Configuring the T1 Environment
- Handling Incoming and Outgoing Calls
- Using Internal Signaling Streams
- Testing the T1 Setup
- Troubleshooting

Understanding T1 Trunks

T1 trunks provide digital communications in North America, Japan, and Hong Kong. A T1 trunk carries 24, 64 Kb/s lines. An additional 8 Kb/s signal provides synchronization information. The combination of the 24 lines plus the synchronization information yields a 1.544 Mb/s signal:

(24 X 64 Kb/s) + 8 Kb/s = 1,544 Kb/s or 1.544 Mb/s

The T1 board provides two 24-line T1 trunks for a total of 48 T1 lines. The T1 board only provides T1 line resources; RDSP or Vantage series boards provide voice processing resources through the MVIP bus.

Transmitting Digital Data

Digital data is composed of zeros and ones that the CO and CPE transmit as an electrical signal. The waveform used to indicate zeros and ones is called line coding. The CO and CPE must use the same line coding method in order to communicate properly. There are three basic types of line codes: Binary, Polar, and Bipolar.

- The Binary signal, also called unipolar, uses two voltage levels to represent zero and one bits. In Figure 6, 0V represents zero and +3V represents one. This line coding method is called unipolar because it is not symmetrical around 0V.
- The Polar signal uses the positive and negative voltage of a certain value to represent zero and one. In Figure 6, –1.5V represents zero and +1.5V represents one. This code is also called a non-return to zero (NRZ).

  In polar and binary signals, the signal level remains constant throughout the time that the bits are transmitted on the line. For example, if four ones are transmitted, the signal remains at some positive voltage for the duration of the ones transmission.

- The Bipolar signal, also called Alternate Mark Inversion (AMI), uses 0V to represent zero and alternating positive and negative pulses to represent ones. AMI is the default line coding method for T1 boards.

Figure 6 shows the three line coding methods.

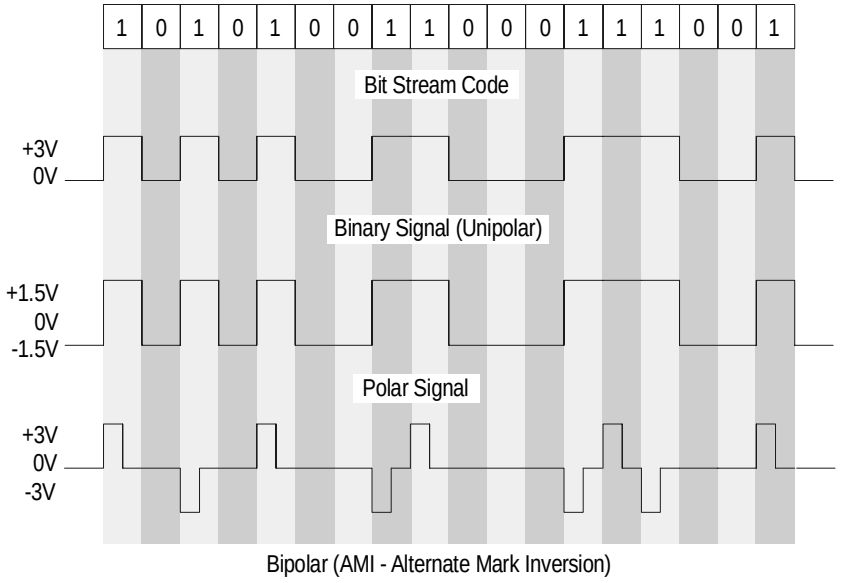

Figure 6. T1 Line Coding Methods

Notice that in AMI line coding, each one alternates polarity and the voltage returns to zero between pulses. Two subsequent ones with the same polarity are called a bipolar violation. Figure 7 shows a proper AMI signal and two bipolar violations. Bipolar violations could lead to crackling on the line. To check for bipolar violations, call RHT_GET_STATUS. See *Troubleshooting* on page 81 for more information about how to handle bipolar violations.

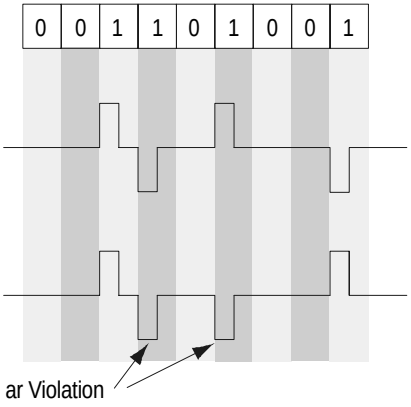

Figure 7. A Bipolar Violation in an AMI Signal

In AMI signaling, a long series of zeros is represented by a constant 0V signal. The two ends can lose timing if there is no signal on the line to synchronize them. To maintain ones density on the line, some carriers replace zeros on the line with ones. T1 devices support two line coding methods for maintaining the appropriate ones density: ZCS and B8ZS.

In ZCS (Zero Count Substitution), the transmitting end replaces the most significant bit of each eight-bit timeslot with a one, regardless of whether the bit was a one or a zero. This method is also called bit stuffing.

In B8ZS (Binary 8-Zero Substitution), the transmitting end replaces a series of eight zeros with a ones-rich pattern. B8ZS coding inserts a bipolar violation as a signal to the receiving end to reinsert the original zeros. Figure 8 shows the B8ZS replacement patterns. Notice that the replacement pattern that is inserted depends on the polarity of the preceding one.

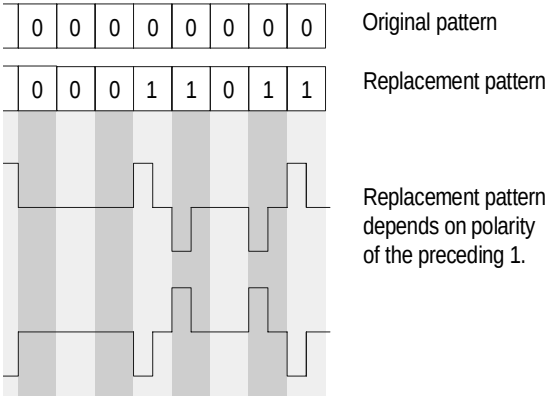

Figure 8. The B8ZS Replacement Pattern

Organizing the T1 Data

Organizing Data into T1 Frames

The T1 line carries data from 24 channels over two pairs of wires (one to transmit data and one to receive it). At the transmitting end, a multiplexer receives a 64 Kb/s signal from each of the 24 channels. It interleaves 8 bits of data from each channel into a single serial stream of data. To carry 24 64 Kb/s channels plus an 8 Kb/s signal, the T1 line runs at 1.544 Mb/s. The process of interleaving data from each channel is called time division multiplexing (TDM).

Figure 9 shows data entering a multiplexer and being transmitted on the line. The multiplexer at the far end separates the data into the original 24 lines.

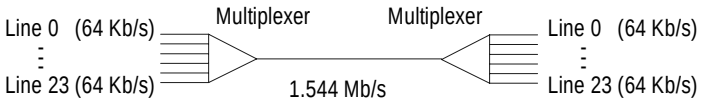

Figure 9. A T1 Line

The transmitting end formats the T1 data into frames so the receiving end can interpret the data. One frame contains 8 bits of data for each channel in 24 8-bit timeslots. The T1 timeslots in a frame are numbered sequentially beginning with 0 (0 through 23). The timeslots correspond to line number, so timeslot 0 carries 8 bits of data for channel 0. A single 64 Kb/s channel (made up of one timeslot from each frame) is called the DS0 signal.

A single bit at the beginning of each frame contains synchronization information. This bit is also called the F-bit or framing bit. Figure 10 shows a single T1 frame with the 24 timeslots and a single F-bit. The 24 64 Kb/s lines plus the F-bit make up a DS1 signal.

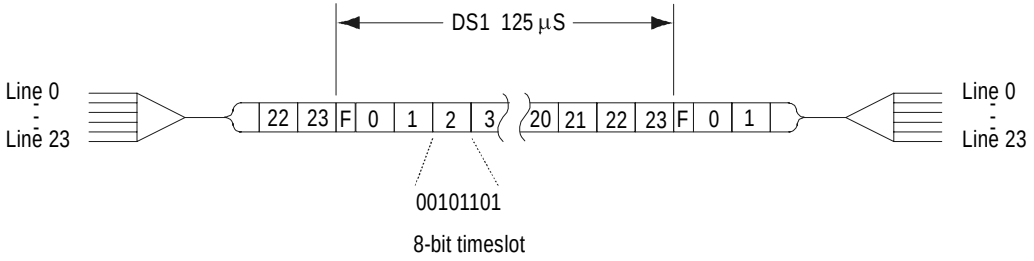

Figure 10. A T1 Frame Carrying 24 8-bit Timeslots

Each frame has a period of 125 µs, or 8000 frames per second. This means that each channel receives 8000 8-bit timeslots per second, or 64 Kb/s:

8000 timeslots/second x 8 bits/timeslot = 64 Kb/s

Organizing Frames into D3/D4 and ESF Superframes

The 24-timeslot frames are grouped into one of two types of superframes: D3/D4 or ESF.

D3/D4 Superframes

D3/D4 superframes consist of 12 frames. The framing bits at the beginning of each frame form the sequence: 100011011100. For example, the F-bit for the first frame is a 1, for the second frame it is a 0, and so on through the 12-bit sequence for the 12 frames, as shown in Figure 11.

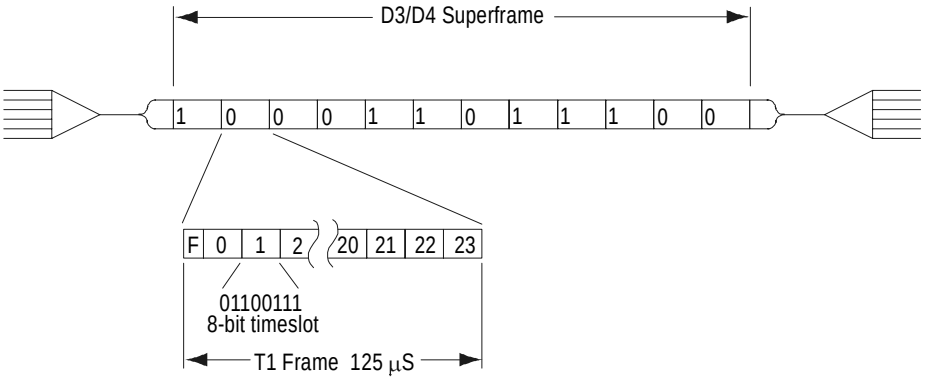

Figure 11. A 12 Frame D3/d4 Superframe Showing the F-bit Sequence

D3/D4 superframes use the least significant bit from each timeslot in the sixth and twelfth frames for signaling. This is called robbed bit signaling, since the signaling bits are essentially robbed from the voice data. In robbed bit signaling, two bits from each channel are robbed per superframe. The first, in the sixth frame, is called the A bit. The second, in the twelfth frame, is called the B bit.

The A and B bit carry information about the line state, such as on hook, off hook, disconnected, or answered. For information about the A and B bit settings in the different protocols, see Appendix A, *T1 Line Protocols,* on page 223.

ESF Superframe

Extended superframes (ESF) consist of 24 frames. Unlike D3/D4 frames, the F-bits are not all used for synchronization. Only the F-bits for frames 3, 7, 11, 15, 19, and 23 carry synchronization information. These follow the pattern: 001011. For example, the F-bit for the fourth frame is a 0, and for the eighth frame it is a 0, and for twelfth frame it is a 1, and so on through the sequence, as shown in Figure 12.

The F-bit for frames 1, 5, 13, 17, and 21 carries error checking information (CRC). For each superframe, the transmitting end calculates a cyclic redundancy checksum for the data and places the results in the following superframe. The remaining frames carry network information.

As with D3/D4 superframes, ESF superframes rob the least significant bit for each timeslot of every sixth frame for signaling information. With 24 frames in a superframe, this means that four bits per channel are used for signaling, as shown in Figure 12. The first is bit A, the second bit B, the third bit C, and the fourth is bit D. In most signaling protocols, C is set to equal A, and D is set to equal B. However, in proprietary protocols, these bits can be set to different values to indicate line state.

Figure 12 shows a single timeslot within an ESF superframe. Each timeslot in the superframe follows the same pattern for signaling bits.

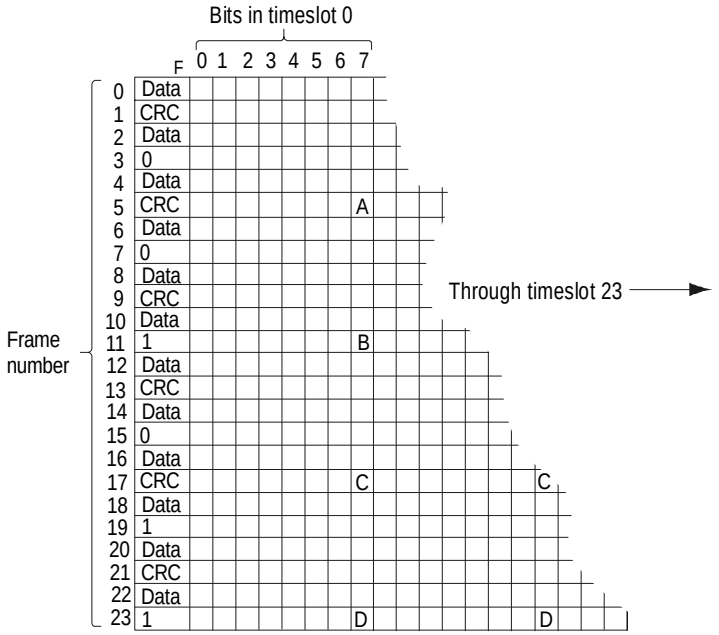

Figure 12. An ESF Superframe

T1 devices support both D3/D4 and ESF framing. Be sure your application uses the same framing method as your carrier. Using the wrong framing method causes the two ends not to synchronize.

Configuring the T1 Environment

Configuring the T1 environment involves the following steps:

1. Setting the clock
2. Loading the line protocol
3. Configuring the carrier

Setting the Clock

T1and E1 boards must have their clock synchronized with the network and with other boards connected in a CT bus. In a CT bus, one board drives the clock and all other boards retrieve the clock from the bus. If the system contains T1 or E1 boards, one drives the clock based on the signal it receives from its network trunk A or B. This synchronizes the CT bus with the T1 or E1 network. Otherwise, an RTNI-ATSI or Vantage PCI board uses its internal oscillator to set the CT bus clock.

Configure the clock using CONFIG_CLOCK. For specific information about setting the clock for boards in an CT bus, examples of system configurations, and sample code, see *Configuring the MVIP-90 Clock* on page 150.

If you do not set the clock correctly or if you specify more than one board to drive the clock, you could either lose data or transfer data improperly to other boards in the CT bus. An incorrect clock setting can also lead to crackling noise on the line.

Loading the Line Protocol

The line protocols determine how the central office (CO) and customer premise equipment (CPE) communicate using the A and B signaling bits. Both ends must use the same protocol in order to communicate properly. The most common protocols are Wink Start, Double Wink Start, Immediate Start, Loop Start, and Ground Start.

The protocol files convert the API functions such as RHT_ON_HOOK and RHT_OFF_HOOK into appropriate signaling patterns and timing for the protocol. For example, if an application calls RHT_ON_HOOK, the protocol file handles any signaling or timing involved in going on hook for the current protocol. An application can load a protocol at run time, so one application can use different protocols without modifying any code. Be sure to load the protocol before configuring the line carrier.

Specify the protocol your application uses either when you configure drivers with the Configuration Wizard or by calling RHT_LOAD_PROTOCOL. The T1 driver uses the Wink Start protocol by default. The two T1 trunks on an T1 board can each run different protocols, but all lines on a specific trunk use the same protocol. To find out what protocol file is currently loaded on a given trunk, call RHT_GET_STATUS. This function also provides the file version, source file name, and compiler version used to generate the protocol.

Once loaded, a protocol stays in effect until it is overwritten by a new protocol or the drivers stop. An application should always load the appropriate protocol at runtime to prevent a previously loaded protocol from causing the application to malfunction.

Table 11 shows the protocols Brooktrout supports and the protocol file names. These files are distributed with the T1 drivers. For specific information about the A and B bit patterns in the T1 line protocols, see Appendix A, *T1 Line Protocols,* on page 223.

Table 11. Protocol File Names

Protocol	File name
Wink Start	wink.mto
Double Wink Start	wink.mto (same as wink start)
Immediate Start	imdt.mto
Loop Start	stlpst.mto and colpst.mto
Ground Start	stgndst.mto and cogndst.mto

The Immediate Start, Wink Start, and Double Wink Start protocols are symmetrical. The CO and CPE transmit identical bit patterns for each line state. Your application uses the same protocol file whether it functions as a CPE or emulates a CO.

The Loop Start and Ground Start protocols are asymmetrical. The CO and CPE transmit a different bit pattern to indicate the same condition. For example, the bit pattern the CO transmits to indicate answer is different from the bit pattern the CPE transmits to indicate answer. If you use these protocols, you load a different protocol file depending on whether your application acts as a CPE or CO.

If your system is in a loopback configuration using a asymmetrical protocols, one trunk uses the CO protocol file and the other trunk uses the CPE protocol file.

Example 2 shows how to load the Wink Start protocol using RHT_LOAD_PROTOCOL.

Example 2. RHT_LOAD_PROTOCOL Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{

   BRKT_HANDLE BoardHandle;
   FILE* ProtocolFile;
   BOOLEAN IoctlResult;
   BRKT_SIZE_T BytesReturned;
   BRKT_SIZE_T FileSize, BytesRead;
   USHORT* pProtocolBuffer;
   struct LoadProtocol_s Protocol;

   BoardHandle = BrktOpenDevice (BRKT_DEVICE_T1_BOARD, 0);

     /* Open Protocol File */
   ProtocolFile = fopen ("WINK.MTO", "rb");

   FileSize = _filelength (_fileno (ProtocolFile);
   pProtocolBuffer = malloc (FileSize);

   if (pProtocolBuffer == NULL)
   {
         printf ("Can't allocate memory for protocol.\n");
         return;
   }

   BytesRead = fread (pProtocolBuffer, sizeof (UCHAR), FileSize,
                      ProtocolFile);
   fclose (ProtocolFile);
```

Example 2. RHT_LOAD_PROTOCOL Sample Code (Continued)

```
   memset (&Protocol, 0, sizeof (Protocol));
   Protocol.Length = FileSize;
   Protocol.Protocol = pProtocolBuffer;
   Protocol.Trunk = M_TRUNK0;            /* Trunk 0 */

   IoctlResult = BrktDeviceIoControl (
         BoardHandle,
         RHT_LOAD_PROTOCOL,
         &Protocol,                      /* Buffer to driver */
         sizeof(Protocol),               /* Length */
         NULL,                           /* Buffer from driver */
         0,                              /* Length */
         &BytesReturned,
         NULL);                          /* Wait till I/O complete */

   if (!IoctlResult)
         printf ("LOADPR failed: BrktGetLastError = %d\n",
               BrktGetLastError (BoardHandle));
   else
         printf ("LOADPR done \n");
   BrktCloseDevice(BoardHandle);

}
```

Configuring the Carrier

The carrier parameters determine what framing methods, line coding methods, and other parameters the board uses to send and receive data. The parameters you set must match those of the carrier.

There are several ways to configure carrier parameters. Configure all channels on a given trunk to use the same values, using CONFIG_CARRIER. To configure the robbed bit, invert, or loopback parameters separately for individual channels, use CONFIG_CHANNEL and RHT_CONFIG_CHANNEL. However, since there is rarely a need to configure these parameters separately, we recommend using CONFIG_CARRIER. To configure only line coding and framing for all channels on a given trunk, use the Configuration Wizard.

For all parameters, contact your carrier for information about the appropriate settings.

Configuring the Framing Method

T1 boards support D3/D4 and ESF framing, as described in *Organizing the T1 Data* on page 42. Setting the incorrect framing mode can cause a loss of synchronization, so be sure your configuration matches your carrier.

Configuring the Line Coding Method

T1 boards support AMI, AMI with ZCS, or AMI with B8ZS line coding, as described in *Organizing the T1 Data* on page 42. Setting the incorrect line coding method can cause bipolar violations or noise on the line, so be sure your configuration matches your carrier.

Configuring Debounce

When debouncing is enabled, the board waits for the signaling bits to be stable before relaying the information to the drivers. Waiting for the bits to stabilize prevents errors from being handled as if they were valid signals. Debouncing, which is also called deglitching, should always be enabled.

The 6 to 9 ms wait is less than the minimum time used for signal recognition protocols, so enabling debouncing should not decrease performance. In fact, it should increase performance, despite the delay, since the driver does not have to handle spurious signals. The driver adds an additional debouncing time at a higher software layer, which further reduces the risk of spurious signals reaching the application.

Configuring the Loopback Mode

There are two software loopbacks set using CONFIG_CARRIER: remote and local. These are used to test a board or the data link. For information about additional hardware loopbacks, see *Troubleshooting* on page 81.

- In a remote loopback, the board immediately transmits all data it receives back to the CO. If all connections are working, the CO should receive the same information it transmitted. If any transmission errors disappear in the remote loopback configuration, the problem is with the clock or carrier settings in your application. If the problem does not disappear, the problem is either the cabling or at the carrier end.
- In a local loopback, the board receives the same data it transmits, without passing through any cabling. If placing your board in a local loopback fixes problems you experienced on the line, the problem is with the cabling or with the carrier. If a local loopback does not fix errors, then the board is not sending valid data.

When you first load the driver, it has both loopback modes enabled by default. In this configuration, both the board and the CO only receive the data they transmit. This configuration lets you verify that both the board and CO can send and receive valid data. For normal operation, remove both loopbacks so that the board and CO can transmit and receive data.

Configuring the Line State

When loaded, each protocol automatically transmits the bit pattern corresponding to the idle state. The drivers ignore the 'Hook' field in the carrier parameters.

Transmitting the proper line state is one reason to load the protocols before configuring the carrier parameters. If you configure the carrier first, the board might transmit an invalid hook pattern on the line. This invalid signal could trigger alarms at the CO. Although the alarms would disappear when you load the line protocol and begin transmitting the appropriate bit pattern, the CO switch might have blocked the line in response to the alarm.

Configuring Invert

The invert parameter sets whether to invert the polarity of data sent and received on the trunks. Invert should be always set to 0 (don't invert) unless technical support explicitly requests that you set it to invert the data.

Configuring Robbed Bit Signaling

Robbed bit signaling should always be enabled if your T1 trunk is connected to a CO. In this mode, your board and the CO use the A and B bits to signal information about the line as described in *Organizing the T1 Data* on page 42. Disable robbed bit signaling only if there is another way to communicate signaling information. Normally, you would only disable robbed bit signaling if the T1 trunk links different company premises using a proprietary protocol.

Example 3 shows how to configure the first trunk of the first T1 board for D3/D4 framing, B8ZS coding, and default values for all other fields.

Example 3. CONFIG_CARRIER Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{

   BRKT_HANDLE BoardHandle;        /* T1 board device handle */
   BOOLEAN IoctlResult;            /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;      /* Bytes returned from IOCTL call*/
   struct RTNI_T1carrierParam_s Carrier;

     /* Open T1 board device */
   BoardHandle = BrktOpenDevice (BRKT_DEVICE_T1_BOARD, 0);

     /* Configure T1 carrier */
     /* Set up same parameters for both trunks */
   memset (&Carrier, 0, sizeof (Carrier));
   Carrier.Size = sizeof(struct RTNI_T1carrierParam_s;
   Carrier.Trunk = M_ALL_TRUNK;
   Carrier.Frame = DT_D4;          /* D3/D4 framing */
   Carrier.Code = DT_AMI;          /* Coding method: AMI with no ZCS */
   Carrier.Debounce = 1;           /* Enable debounce (deglitch) */
   Carrier.Loopback = 0;           /* Disable loopback */
   Carrier.Invert = 0;             /* Normal polarity */
   Carrier.RobbedBit = 1;          /* Enable robbed bit signaling */
```

Example 3. CONFIG_CARRIER Sample Code (Continued)

```
IoctlResult = BrktDeviceIoControl (
      BoardHandle,
      CONFIG_CARRIER,
      &Carrier,                /* Buffer to driver */
      sizeof(Carrier),
      NULL,
      0,
      &BytesReturned,
      NULL);                   /* Wait until I/O is complete */
if (!IoctlResult)
      printf ("CONFIG_CARRIER failed: BrktGetLastError = %d\n",
            BrktGetLastError (BoardHandle));


BrktCloseDevice(BoardHandle);
Return(0);
```

Handling Incoming and Outgoing Calls

Processing Calls

Call processing involves setting up and tearing down calls. Calls have to be processed before they are connected, so be sure to verify call processing in the early stages of development.

Table 12 shows the functions used in sending and receiving calls over a T1 line. These functions are the same as those used in analog telephony. The protocol files translate these functions into the specific bit patterns and handshaking signals used by the T1 protocols, as described in *Loading the Line Protocol* on page 48.

Table 12. API Functions Used in T1 Signaling

Function	Description
RHT_WAIT_LINE_ON	Waits for an incoming call and automatically acknowledges it.
RHT_OFF_HOOK RHT_SEIZE_LINE	Answers an incoming call or initiates an outgoing call.
RHT_ON_HOOK RHT_DISCONNECT	Terminates incoming or outgoing calls.
RHT_STOP	Terminates a function.
RHT_WAIT_LINE_OFF	Monitors the line waiting for a disconnect.
RHT_WAIT_ANSWER	Monitors the line waiting for an answer.
RHT_WAIT_IDLE	Waits for an idle pattern on the line.

Handling Incoming Calls

Handling incoming calls involves the following steps:

- Detect an incoming call
- Detect digits
- Send control tones
- Answer the call
- Monitor for a disconnect
- Terminate the call

Detecting an Incoming Call

The application calls RHT_WAIT_LINE_ON when it is ready to receive a call. This function waits until it receives a seizure or a ring, then sends any acknowledgment signals required by the protocol. After acknowledging the call, if required, the function waits for an amount of time specified by the RDG_*LOCAL_ACK_GUARD_TIME* parameter, then returns. The application is then ready to start receiving digits. The series of events involved in detecting an incoming call is shown in Table 13.

Table 13. RHT_WAIT_LINE_ON Sequence

Action	Duration
Receive seizure or ring	
Send acknowledgment (if required)	
Wait	*RDG_LOCAL_ACK_GUARD_TIME*
Return	

The function automatically sends any acknowledgments required by the protocol. This serves two purposes. First, it isolates the application from the protocol. You can communicate with different carriers simply by switching protocol files rather than making changes to the application. Second, it improves timing. In a heavily loaded system, threads and processes can be preempted at almost any time, and it might take several seconds before they can run again. If one function detected a call and another sent the acknowledgment, the two signals could be several seconds apart. This delay might exceed the maximum allowed by the originating end.

The guard time adds a space between two consecutive line function calls so the remote end can detect a change in the signaling bits. One example of when a guard time is necessary is in an application designed to answer a call, play a file, then disconnect. If the application cannot play the file, it immediately hangs up. If the answer is on the line for a very short amount of time before the disconnect, the remote end might mistake the on hook/off hook/on hook pattern as a wink rather than an answer followed by a disconnect. A guard time after going on hook guarantees that the signal was on the line long enough for the remote end to recognize the signal. Although a guard time is not necessary in RHT_WAIT_LINE_ON, it is available for consistency. It is set to 0 by default.

If the application calls RHT_WAIT_LINE_ON when the line is in a state other than idle, the function returns an error.

RHT_WAIT_LINE_ON should continuously wait for a call, even though you can limit the function's run time with the parameter MDP_WAIT_LINE_ON_TIMEOUT. If you use this timeout to monitor for other conditions, you could miss incoming calls. Use another thread to do any necessary monitoring. To terminate RHT_WAIT_LINE_ON, use RHT_STOP. The application is notified that the function terminated and will take any appropriate action.

Detecting Digits

Once the application acknowledges the incoming call, the CO usually sends call setup digits. In T1 applications, the CO generally sends call setup information using MF or DTMF digits, although Brooktrout boards with voice processing capabilities recognize rotary, DTMF, MF, or R2 digits.

Before detecting new digits from an incoming call, flush the digit buffer using RHT_FLUSH_DIGIT. Otherwise, digits stored in the buffer from a previous call could be handled in the new call.

The best time to flush digits is after going on hook but before calling RHT_WAIT_LINE_ON. If you flush digits after calling RHT_WAIT_LINE_ON, your thread could get pre-empted and rescheduled for after the application has started receiving digits from a new call. When the application eventually flushes the digit buffer, it clears digits from the current call and loses that data.

For more information about handling digits, see Chapter 2, *Digit Handling,* on page 15.

Sending Control Tones

In the Wink Start, Double Wink Start, or Immediate Start protocols, your application emulates both the receiving end CO and subscriber. The CO portion sends control tones such as ringback and busy to indicate the status of the call to the other end. In Loop Start or Ground Start protocols, your system acts only as a subscriber and does not send control tones.

When a subscriber receives a call, the CO emulation part of the application should determine the status of the line and send the appropriate line signaling bits and either a busy or ringback tone back to the caller. It also sends the hangup tone when the subscriber initiates the call disconnection. Many applications skip sending control tones and instead send the answer signal immediately after receiving the incoming call, leaving the line silent until the application is ready to start playing prompts or a person can handle the call. If a calling party does not hear control tones, they might think the call was lost and hang up, especially if the application had to perform a lengthy task such as querying a data base or checking the status of agents at a call center.

Sending control tones also improves the timing of the call if there are many transit COs between the calling party and your application. In this case, it could take several hundred milliseconds to establish an audio path between the two ends. If the application immediately begins playing audio prompts after receiving a call, the caller could lose the first portion of the prompt that was played while the audio path was being established. Sending control tones establishes a path before the application begins playing so the caller hears the entire audio prompt.

Answering Calls

Answer calls detected by RHT_WAIT_LINE_ON using RHT_OFF_HOOK or RHT_SEIZE_LINE. Only the underlined functions are interchangeable, so all discussion of RHT_OFF_HOOK also applies to RHT_SEIZE_LINE.

When RHT_OFF_HOOK answers a call, it automatically transmits the appropriate answer supervision signal towards the switch directly connected to your system. The switch relays this information back to the preceding switching equipment and so on until the information reaches the originating equipment. This serves as a signal to the originating office to start billing the caller.

After transmitting the off hook or answer supervision bit pattern, RHT_OFF_HOOK waits for a duration specified by *RDG_LOCAL_ANSWER_GUARD_TIME* before returning, as shown in Table 14.

Table 14. RHT_OFF_HOOK Sequence

Action	Duration
Transmit answer signal	
Wait	*RDG_LOCAL_ANSWER_GUARD_TIME*
Return	

The guard time is important in case your application needs to hang up immediately after answering a call. Without a guard time, the off hook pattern would be on the line for a very short period of time. The CO could misinterpret the resulting on hook/off hook/on hook pattern as a wink or some other error condition and not recognize that the line is idle. The CO continues waiting for your application to answer while your application waits for a new incoming call. This deadlock continues until the CO times out (which might not happen) or you reset your system.

In Loop Start or Ground Start protocols, audio connection is not established until you call RHT_OFF_HOOK. If the application fails to answer the call properly, the caller does not hear any files the application plays.

In Wink Start, Double Wink Start, and Immediate start protocols, the audio path is established soon after the CO sends digits so the remote end can hear the control tones. With the audio path established, the remote end will also hear any files the application plays even if the application fails to call RHT_OFF_HOOK. If the CO does not receive the answer signal, however, it times out within two to four minutes and terminates the call. To the application and the caller, it appears that the call was being handled properly until the line suddenly disconnected. If your application reports that it received a disconnect after two to four minutes while it was processing the call, check to be sure the application called RHT_OFF_HOOK to answer the call.

If you are sure that your application called RHT_OFF_HOOK, consider the other extreme. The application might call RHT_OFF_HOOK too soon after it detects digits. In this case the switches down the path might not be ready to receive the answer signal, so they do not detect it. The application should send control tones after receiving digits so the COs in the path are ready to receive the answer signal.

Monitoring for Disconnect

Whether you received or originated the call, you should continuously monitor the line for disconnect using RHT_WAIT_LINE_OFF. The sooner you detect the disconnect, the sooner the line is free for new calls, allowing you to reach more people in the same amount of time without increasing the number of lines. RHT_WAIT_LINE_OFF returns successfully when it detects a disconnect or returns an error if it detects other conditions. If RHT_WAIT_LINE_OFF returns an error, use RHT_GET_STATUS for more information about the termination.

When the application detects a disconnect, it should terminate what it is doing, update some information, and hang up in preparation for a new call. Wink Start, Double Wink Start, and Immediate Start protocols provide reliable disconnect signals. Loop Start and Ground Start protocols usually don't provide disconnect information. In these protocols, only the CO receives answer and disconnect signals. We recommend that you use one of the protocols that provide disconnect information if that protocol is available from your carrier.

If the application is running a voice processing function when it detects disconnect, it should terminate that function immediately. There are two ways to terminate a function. The first uses two threads (or processes) per channel: one to run the line monitoring function and another the run the voice processing function. The main thread synchronizes these two threads.

A better way to terminate VP functions is to run the functions so they terminate automatically when the application detects a disconnect. To do this, set the field 'LineTerm0' in the *VPstartStop_s* or *RhtDialDigit_s* structures. This field provides a way for the VP driver and the T1 driver to communicate.

With ‘LineTerm0’ set, the T1 driver automatically runs RHT_WAIT_LINE_OFF. It continues monitoring the line until the VP driver sends a signal that the VP function terminated or until the T1 driver detects a disconnect. When it detects a disconnect, the T1 driver signals the VP driver to terminate the function. When the VP function terminates, check the condition that caused termination using RHT_GET_STATUS. A T_RHT_LINE_OFF condition means that RHT_WAIT_LINE_OFF caused the function to terminate.

This second approach is easier to implement because it does not involve separate threads. However, RHT_WAIT_LINE_OFF only runs when a VP function is running. If the application spends long periods of time performing non-VP functions, a disconnect would not be reported until the next VP or line function runs. This could keep the line busy longer than necessary. If this delay is not acceptable, then use a separate thread to monitor the line while no VP functions are running.

Since RHT_WAIT_LINE_OFF is an exclusive function, you cannot start any other exclusive functions when you start a VP function with ‘LineTerm0’ set.

Terminating an Inbound Call

Terminate calls using RHT_ON_HOOK or RHT_DISCONNECT. These functions are interchangeable, so all discussion of RHT_DISCONNECT also applies to RHT_ON_HOOK.

RHT_DISCONNECT transmits a disconnect (idle) pattern for the time specified by *RDG_LOCAL_IDLE_DUR*. It then waits for the remote end to disconnect for a time specified by *RDG_REMOTE_IDLE_TIMEOUT*. After it receives the disconnect, RHT_DISCONNECT waits for a time specified by *RDG_LOCAL_IDLE_GUARD_TIME* then returns. Table 15 shows the sequence involved in terminating a call.

Table 15. RHT_DISCONNECT Sequence

Action	Duration
Transmit disconnect	*RDG_LOCAL_IDLE_DUR*
Wait for disconnect	*RDG_REMOTE_IDLE_TIMEOUT*
Guard time	*RDG_LOCAL_IDLE_GUARD_TIME*
Return	

Both *RDG_LOCAL_IDLE_DUR* and *RDG_LOCAL_IDLE_GUARD_TIME* guarantee that the disconnect pattern is present on the line for a certain amount of time.

The remote end only sends an idle pattern when the other party hangs up. If the other person does not know that your application has disconnected, it might take a while for them to hang up. An application should play a busy signal when it hangs up so the other party knows you have disconnected.

If the remote end disconnects first and your application calls RHT_DISCONNECT in response to their disconnect signal, it will return almost immediately. The only delays are those caused by *RDG_LOCAL_IDLE_DUR* and *RDG_LOCAL_IDLE_GUARD_TIME*.

If your end disconnects first, RHT_DISCONNECT waits for the other party to disconnect for a time specified by *RDG_REMOTE_TIMEOUT*, which is usually infinite. However, the CO the other party is connected to usually times out in one to four minutes if the other party does not disconnect. When it times out, the CO transmits a disconnect signal on its own. If the CO does not have a timer, it does not transmit a disconnect and your application must wait for the other party to hang up.

If RHT_DISCONNECT does not return within a few seconds, it is probably because the other party has not hung up rather than a problem with the application. Call RHT_GET_STATUS to see the signaling bits currently present on the line if you think the function is taking too long to return. For most protocols, if the A and B bits are both 1, the other party has not yet hung up. When the line is idle, both bits are usually set to 0.

The reason RHT_DISCONNECT waits for a disconnect before returning is so the application knows when the line is free. If RHT_DISCONNECT returned immediately, the application would not know when the line could be used for new calls. The application would have to call RHT_WAIT_IDLE to monitor the line for disconnect after RHT_DISCONNECT returns.

You can change *RDG_REMOTE_IDLE_TIMEOUT* to be less than infinite, but this does not free the line any faster. If RHT_DISCONNECT does not receive the disconnect signal within a time specified by *RDG_REMOTE_IDLE_TIMEOUT*, the function returns an error and the application resumes executing. However, the application cannot make or receive another call until the line is idle, so you do not gain anything by regaining control. Any exclusive line functions also return an error until the line becomes idle. The only way the application knows then the line becomes free is by calling RHT_WAIT_IDLE. This function returns when the line becomes idle, and the application can proceed. Since RHT_WAIT_IDLE is built into RHT_DISCONNECT, it is most efficient to wait for RHT_DISCONNECT to detect the idle and return.

Handling Outbound Calls

Handling outbound calls involves the following steps:

- Seize a line
- Dial out
- Monitor the line for answer
- Process the call
- Terminate the call

Seizing a Line

Seize an idle line using either RHT_SEIZE_LINE or RHT_OFF_HOOK. These functions are interchangeable, so all discussion of RHT_SEIZE_LINE also applies to RHT_OFF_HOOK.

RHT_SEIZE_LINE transmits the line seize signal and waits for an acknowledgment from the remote end for a time defined by *RDG_REMOTE_ACK_TIMEOUT*. It then waits for a time determined by *RDG_LOCAL_SEIZE_GUARD_TIME* and returns. If RHT_SEIZE_LINE does not receive an acknowledgment, it returns the hook to idle and returns an error. Table 16 shows the sequence of seizing a line.

Table 16. RHT_SEIZE_LINE Sequence

Action	Duration
Transmit line seize	
Wait for acknowledgment	*RDG_REMOTE_ACK_TIMEOUT*
Wait	*RDG_LOCAL_SEIZE_GUARD_TIME*
Return	

Only Wink Start and Double Wink Start protocols require a wink to acknowledge the line seizure. A wink is on hook/off hook/on hook sequence where the duration of the off hook must be within a range defined by *RDG_REMOTE_MIN_WINK* and *RDG_REMOTE_MAX_WINK*. This wink must be sent within a time specified by *RDG_REMOTE_ACK_TIMEOUT* after the seizure.

If RHT_SEIZE_LINE does not receive an appropriate wink, it abandons the call, sends an idle pattern, and returns an error. If RHT_SEIZE_LINE repeatedly returns an error indicating that the wink received was out of specification, the wink parameters are probably not set correctly.

In Immediate, Loop and Ground Start protocols, no acknowledgment is necessary.

Glare Resolution

If the line is not idle when the application calls RHT_SEIZE_LINE, BrktGetLastError( ) returns BRKT_ERROR_CODE(IO_DEVICE). See *Troubleshooting* on page 81 for more information on how to handle the error.

It is possible for the application to try to seize the line as the CO tries to send a call. In this situation, called a glare, RHT_SEIZE_LINE returns an error but does not set the line back to idle. In order to accept the incoming call, the application calls RHT_OFF_HOOK. If it does not accept the call, it calls RHT_DISCONNECT to set the line back to idle. Check with your carrier to see how they want applications to handle glare situations (also called glare resolution).

Dialing Out

After the application seizes the line, it dials the appropriate number using RHT_DIAL. It can dial out using either MF or DTMF tones. While dialing, the application monitors the line by setting field in the *RhtDialDigit_s* structure as described in *Monitoring for Disconnect* on page 64.

If the function returns successfully, the application monitors the call for answer. If the function returns T_RHT_LINEOFF, then the function terminated because of a line condition. Call RHT_GET_STATUS to find out what line condition occurred. Other codes indicate more serious error conditions. You should abandon the call using RHT_DISCONNECT and try the call again.

For more information about how T1 applications dial digits, see Chapter 2, *Digit Handling,* on page 15.

Monitoring for Call Answer

To detect the answer supervision signal in Wink Start, Double Wink Start, or Immediate Start protocols, use RHT_WAIT_ANSWER. RHT_WAIT_ANSWER monitors for the answer bit pattern and informs the application when the remote end answers.

RHT_WAIT_ANSWER returns successfully if it detects an answer pattern or returns an error if it detects any other pattern such as a disconnect or bit errors. If RHT_WAIT_ANSWER does not detect a bit change, it continues running until the application terminates it. In Ground Start and Loop Start protocols, which do not return an answer supervision signal, the function runs forever or until an error occurs.

In some protocols, the bit patterns are the same when dialing out and when the remote end sends a disconnect signal. To distinguish between these, call RHT_WAIT_ANSWER immediately after dialing out and before calling any other line function. After RHT_WAIT_ANSWER detects the answer, call RHT_WAIT_LINE_OFF to detect disconnect. Calling the functions in this specific order allows the driver to keep track of the call history and to differentiate between ambiguous bit patterns based on their context.

RHT_WAIT_ANSWER only monitors for answer based on the answer supervision signal. If the application does not use a protocol that sends the answer supervision signal, such as Loop Start or Ground Start, or if the application needs more information about the call than whether it was answered, use RHT_START_PCPM. RHT_START_PCPM accesses the Programmable Call Progress Monitoring (PCPM) algorithm that runs on the board's DSP. PCPM monitors the line for control tones, voice, or silence on the line and determines the status of the call. Since PCPM monitors all audio on the line, it can provide information about busy, no answer, or other line conditions.

Since PCPM uses only sound or silence to determine the call status, it cannot be absolutely accurate in detecting answer. For example, if a busy switch plays a recording saying that all circuits are busy, the PCPM algorithm might recognize the human voice and determine that the call has been answered. For this reason, you should run RHT_WAIT_ANSWER and RHT_START_PCPM simultaneously. RHT_WAIT_ANSWER provides accurate answer detection while RHT_START_PCPM provides other information about the status of the call.

If the application starts RHT_START_PCPM with *VPstartStop_s.LineTerm0* set, the VP driver automatically requests that the T1 driver run RHT_WAIT_LINE_OFF. RHT_WAIT_LINE_OFF monitors the line for a disconnect, but it also detects answer. When RHT_WAIT_LINE_OFF detects an answer or disconnect, it terminates RHT_START_PCPM. The application then calls RHT_GET_STATUS to determine the status of the line.

Example 4 shows the steps involved in using RHT_START_PCPM to detect answer.

Example 4. Monitoring for Answer

1. The application starts RHT_START_PCPM with *VPstartStop_s* fields 'LineTerm0' and 'Timeout' set.
2. When RHT_START_PCPM returns, the application calls RHT_GET_STATUS for the VP device and checks *VPchanStatus_s.TermType*.
3. If *VPchanStatus_s.TermType* contains T_RHT_PCPM, then RHT_START_PCPM determined the line state. The field *VPchanStatus_s.PCPMtype* contains the line status information.
4. If *VPchanStatus_s.TermType* contains T_RHT_TIMEOUT, then RHT_START_PCPM did not detect any condition within the allotted amount of time.
5. If *VPchanStatus_s.TermType* contains T_RHT_LINEOFF, the CO either answered the call or disconnected before answering the call. The application calls RHT_GET_STATUS for the line device and checks *RTNI_lineStatus_s.TermType*. This field contains information about the status of the line.

Terminating an Outbound Call

Terminating an outbound call uses exactly the same procedure as terminating an inbound call. For more information, see *Terminating an Inbound Call* on page 66.

Transferring Calls

You can transfer calls on digital lines if the equipment connected to your system supports call transfers. Most COs do not support call transfer if your application is emulating another CO. However, if your application is emulating customer equipment, the CO might allow call transfers if you have the service enabled. In most cases, you can do call transfers if you have Centrex lines or if your system is connected to a PBX that supports digital lines as extensions. You will need support from your PBX manufacturer, but you can transfer calls to any destination your PBX supports. Loop Start and Ground Start protocols are the only two that normally do call transfers.

Use RHT_HOOK_FLASH to transfer calls. This is the same function used in analog lines. When you call RHT_HOOK_FLASH, the driver uses the protocol file to send the appropriate bit patterns and timings for the digital hook flash.

To perform a hook flash, the channel must be off hook and transmitting the appropriated bit pattern for that protocol. A brief on hook pattern is transmitted for the duration specified by *RDG_LOCAL_FLASH_DUR*, followed by the off hook pattern again for the duration specified by *RDG_LOCAL_FLASH_GUARD_TIME*.

The only requirements for a protocol to support a hook flash are on hook and off hook bit patterns that differ from each other. However, the flash signal might be ignored or interpreted as a disconnect followed by a line seizure if the CO does not support call transfer. The only protocols that cannot transmit a flash hook signal are the CO versions of Loop Start and Ground Start.

Sending and Detecting Winks

The Wink Start and Double Wink Start protocols use wink signals as part of their call setup process. In order to perform a wink, the channel must be in the on hook state, transmitting the appropriate bit pattern for that protocol. An off hook pattern is transmitted for the duration specified by *RDG_LOCAL_WINK_DUR*, followed by the on hook pattern for the duration specified by *RDG_LOCAL_WINK_GUARD_TIME*.

In the Wink Start protocol, the driver uses the protocol file to automatically send or detect the wink that acknowledges line seizure. In the Double Wink Start protocol, however, a second wink must be sent by the receiving end after receiving all digits and before answering. Since the timing of this second wink is determined by when the application finishes receiving digits and not by the protocol, the driver cannot send the wink automatically. Use the RHT_SEND_WINK and RHT_WAIT_WINK functions to send or receive this second wink in the Double Wink Start protocol. Add a pause after the second wink in the Double Wink Start protocol so the CO can distinguish between the wink and the answer signal. Check with your carrier provider for the duration of the pause.

In order to detect a wink, the channel detects an off hook pattern on the line for a period of time within a range defined by *RDG_REMOTE_MIN_WINK* and *RDG_REMOTE_MAX_WINK*. RHT_WAIT_WINK returns an error if the wink duration does not match the specifications. Unlike the wink detection performed when seizing a line, RHT_WAIT_WINK does not set a maximum time for the wink to be received. It waits until it detects a wink, a signaling error occurs, or the function is stopped externally.

If the application needs even more control over a Wink Start or Double Wink Start protocol, it can load the Immediate Start protocol file instead and manage sending and receiving all winks manually.

Using Internal Signaling Streams

The T1 board has internal streams that hold data received on the line. The board holds data for lines 0 through 23 in internal streams 16 and 18, timeslots 0 through 23. It holds signaling bits (ABCD) for lines 0 through 23 in internal timeslots 17 and 19, timeslots 0 through 23.

Access the signaling bits using the SET_OUTPUT and SAMPLE_INPUT MVIP functions. Applications can use these functions to set and retrieve signaling bits directly from the hardware instead of relying on higher level driver functions such as RHT_GET_LINE and RHT_GET_STATUS. Access to low level signaling bits should only be used for tracing purposes, since it can interfere with normal driver functions such as RHT_ON_HOOK.

To write bits to a specific internal stream/timeslot, put the timeslot in message mode and use SET_OUTPUT. The hardware takes the bits you write to the signaling streams and sends them as signaling bits. To read bits from a specific stream/timeslot, use SAMPLE_INPUT. See the *RealCT Direct API Reference Manual* for more information about SET_OUTPUT and SAMPLE_INPUT.

For information about switching data received in streams 16 and 18 over the CT bus, see Chapter 5, *MVIP-90,* on page 143, and Chapter 6, *MVIP-95,* on page 183.

Testing the T1 Setup

Testing the Installation

Use the samples that came with your software to test your boards, drivers, and installation before you write your application. You can use the provided source code to help develop your application. The samples are located in subdirectory \RHT\SAMP\T1.

The most useful samples are:

T1INIT	Sets the clock, initializes carrier parameters and checks carrier status. Older code samples set the clock reference to be Trunk A, and configured the carrier for D4 framing and AMI line coding. If your installation requires different values, make sure to alter the samples accordingly. Newer samples configure most parameters using the command line. The defaults are the same as earlier samples.
WAITRING [line]	Waits for a call on the specified line (0 by default).
OFFHOOK [line]	Answers an inbound call or initiates an outbound call on the specified line (0 by default).
ONHOOK [line]	Disconnects a call on the specified line (0 by default).
WAITANS [line]	Waits for the remote end to answer on the specified line (0 by default).
WAITOFF [line]	Monitors for a disconnect on the specified line (0 by default).
LSB [line]	Displays all relevant information about a line (structure *RTNI_lineStatus_s*).
PLAYT [line] [file]	Plays a file while monitoring for a disconnect on the line (default: line 0, file test.vox).
DIAL [line] [digits]	Dials digits while monitoring for a disconnect on the line. Both command arguments are necessary.
BSTAT	Queries carrier status.
BINFO	Displays board and driver information.

CONN <output stream> <output timeslot> <input stream> <input timeslot>	Makes MVIP connections.
QC <output stream> <output timeslot>	Queries MVIP connections.
DIGIT [line]	Reads digits sent by the CO (line 0 by default). In subdirectory \RHT\SAMP\STD.
TWAIT [line]	Waits for digits (line 0 by default). It must be modified to monitor a T1 line for disconnect instead of an LS line. In subdirectory \RHT\SAMP\STD.

Run these samples either individually or as a batch file. For example, the following batch file initializes the T1 line, makes the MVIP connection between line 0 and VP 0, receives an inbound call, plays a file, and then disconnects:

```
T1INIT
CONN 16 0 6 1
CONN 6 1 16 0
WAITRING 0
TWAIT 0
OFFHOOK 0
PLAYT 0 TEST.VOX
ONHOOK 0
```

In this example, TWAIT has to be modified to monitor a T1 line rather than an LS line.

Testing the Application

There are four basic ways of testing your application:

- Using AccuSpan
- Placing calls over a live T1 line
- Using a T1 simulator
- Connecting trunks A and B

Using AccuSpan

AccuSpan is the best way to test the overall robustness and fault tolerance of your system. AccuSpan is a DOS-based utility that gives you full control over the carrier configuration, alarms, and bit patterns sent and received. It can also receive or generate calls. By changing the carrier parameters, you can test different scenarios for your application. For example, you can see what kinds of errors you receive if you configure your application to use ESF framing and the CO uses D3/D4. You can also transmit alarm conditions to see how the remote end responds. To use AccuSpan, connect two back to back computers with one running the application and the other running AccuSpan.

The most powerful feature that AccuSpan provides is signaling mode. In signaling mode you can transmit any possible bit pattern to the application, where the received bit pattern is always shown on screen. By sending certain bit patterns at specific times, you can manually generate all events the protocol supports, such as line seizure, answer, or disconnect. You can then send valid events at invalid times, send invalid bit patterns at several different states of the call progress, or send signals that are too long or too short and check how the application responds. Using signaling mode requires a good understanding of the protocols. If you are not already familiar with the protocols, view the bit patterns sent or received on the line during normal operation to see how the protocol operates. Appendix A, *T1 Line Protocols,* on page 223, also provides information about bit patterns used in the line protocols.

Placing Calls over a T1 Line

Testing your application over a live T1 line gives you a controlled environment and requires no special knowledge of the protocols. If the application passes a few tests such as making simultaneous calls or consecutive calls on the same line, the application will probably work for real calls. However, when testing the application over a real T1 line, the telephone company controls the environment so there is little chance to test error conditions. Before testing an application on a live T1 line, you should use AccuSpan to eliminate as many error conditions as possible.

Using a T1 Bulk Call Generator

Using a T1 bulk call generator is a good way to stress-test your application. Most bulk call generators allow at least four trunks, or 96 lines, to place calls simultaneously. However, using a call generator should not replace other tests, particularly tests over a real line. The bulk call generator usually does not provide a way of testing error handling since they follow scripts that you determine and send the correct signals on the line. They might also handle exceptions and timing issues differently than your CO.

Connecting Trunks A and B

When you connect trunks A and B, you receive data on one trunk that you send from the other trunk. Using this configuration, you can test samples that complement your application or use the samples that came with your software. Using either your own or provided samples, you can test inbound and outbound call processing, send events out of order to see how the application reacts, and stress test the application.

When you connect both trunks, be sure that the transmit leads from one cable connect to the receive leads of the other cable, and vice versa. There are two ways to be sure the cables connect properly. The first is to use a crossover cable, which connects the transmit and receive leads of one cable to the appropriate leads on the other cable. The other way (available only with RTNI-2T1 or -2E1 boards) is to set the mode of the trunk using hardware jumpers, then connect the trunks using a regular cable. Set one trunk to user mode and the other trunk to network mode, as described in your hardware installation guide. For normal operation, both trunks should be set to user mode. Figure 13 shows trunks A and B connected by a cable. In this configuration, trunk A and B transmit and receive data from each other rather than the CO.

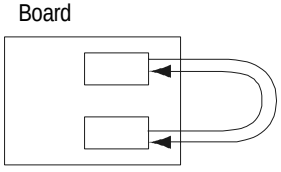

Figure 13. Connecting Trunks A and B in T1

Troubleshooting

If a line function returns an error code, use BrktGetLastError () to determine what caused the error. The two most common errors are lost synchronization or a protocol error.

- In a synchronization error, BrktGetLastError(DeviceHandle) returns BRKT_ERROR_CODE(UNEXP_NET_ERROR)
- In a protocol error, BrktGetLastError(DeviceHandle) returns BRKT_ERROR_CODE(IO_DEVICE)

Handling Synchronization Errors

When a trunk loses synchronization, any bits the trunk receives are invalid. After a loss of synchronization, terminate all calls and check the trunk status using QUERY_CARRIER_STAT. When the trunk is synchronized with the network the application can proceed.

QUERY_CARRIER_STAT returns information about the current alarm and synchronization status of the line in the structure *RTNI_T1carrierStatus_s*. It also returns any errors that have occurred since the last time the application called that function. This information can help you measure the quality of the link to the network. If the data returned indicates the line is not synchronized or that alarms are present and the condition persists for longer than a few hundred milliseconds, the application should terminate all calls and issue an error message.

RTNI_T1carrierStatus_s returns the following information about the line:

Alarm Indicates whether there are any alarms on the line. There are three alarm types:

Red A red alarm indicates that the trunk is not receiving a valid signal or is receiving no signal at all.

Yellow

The CO sends a yellow alarm to indicate that it is not receiving a valid signal from your system. If the T1 board receives a yellow alarm, then the connection from the remote end to the local end is good, and the problem is in the data transmitted to the remote end from the local end.

One reason for a yellow alarm is faulty wiring. The connection uses one pair of wires to transmit data and one pair to receive data. A yellow alarm could happen if the wire pair that transmits data from the board to the CO is broken or disconnected but the other wire is in good condition. To see if the wiring is faulty, set the board in a remote loopback where all data transmitted by the CO is immediately transmitted back to the CO by the board. If all the wiring is good, the CO receives the same data it generates, and the alarm it generates should disappear. In this case, the problem is probably that the board is sending faulty data.

If the CO continues generating a yellow alarm in a remote loopback, the wire could be faulty anywhere from the board to the CO. To identify if the problem is with the network lines or with the customer premise lines, use a CSU loopback. In this mode, all data transmitted by the CO is immediately transmitted back to the CO by the CSU. If the problem disappears, the faulty line lies between the CO and the T1 board. If the problem does not disappear, the problem is with the network lines.

Blue

A blue alarm is a signal containing all ones. If the framing bits are also ones, then the blue alarm is an unframed all ones. If the framing bit follows the appropriate pattern then the signal is a framed all ones.

Either end can send a blue alarm to test the data link. You can also send a blue alarm to test a trunk that cannot send calls. In either direction, when the blue alarm is removed, the alarm should clear and synchronization should be restored. If the alarm clears but the ends are still not synchronized, then your application and the CO are probably using a different framing method. See *Configuring the Carrier* on page 52 for more information on how to set the correct framing method.

Slips

Buffer slips indicate that the T1 board is not reading and writing data to the line at the same rate as the remote end. Usually, an incorrect clock setting or faulty clock source is responsible for buffer slips. For more information about setting the clock, see *Setting the Clock* on page 47.

Bipolar Violation

In AMI line coding, ones alternate positive and negative polarity, as described in *Transmitting Digital Data* on page 38. A bipolar violation occurs when a one is the same polarity as the preceding one.

Bipolar violations indicate that the application and the remote end are not using the same line coding method, or the signal is too weak for the board to detect it properly. Be sure the line coding method your application uses is the same one used by the carrier, as described in *Configuring the Carrier* on page 52. If the line coding method is correct, then the signal might be too weak for the board to detect it. Since the CO can handle weaker signals, the CO might not experience bipolar violations. If the board reports errors and the CO does not, a weak signal is probably the cause.

Having a mismatch in the length of the CSU to board cable and the equalization setting on the board can also lead to bipolar violations. Check the equalization setting on the board as described in your hardware installation guide to be sure they are set for the correct length cable.

Sync

The synchronization field indicates whether or not the network and T1 board are synchronized. If they are not synchronized, it means that the board is not receiving framing information. This can happen if there is no signal on the line, if the board is receiving an invalid or test signal (such as blue alarm), or if the framing mode is not set properly. Be sure that you have set the appropriate clock mode and framing method, as described in *Configuring the T1 Environment* on page 47.

If changing the clock mode or framing method doesn't restore synchronization, the cable might be improperly installed or the CSU is in loopback mode. Use a remote and a local loopback to isolate whether the problem originates at the CO or your system. For more information about loopbacks, see *Configuring the Loopback Mode* on page 53 and *Using Loopbacks* on page 88.

Handling Protocol Errors

A protocol error occurs when the signaling bits read from the line don't match what was expected at that time. For example, a function detects call answer while monitoring for disconnect. An unexpected event is not necessarily an error, since there are times when more than one event could happen. However, the driver returns an error if any event other than the one the function is waiting for takes place.

The best way to recover from a protocol error is to call RHT_DISCONNECT to set the local end to idle, then call RHT_WAIT_IDLE and wait for it to return. When that happens, both ends are in an idle state and are ready to send or receive calls. When the line is idle, the application can continue.

Call RHT_GET_STATUS for more information about what caused the protocol error. RHT_GET_STATUS returns information about the protocol and compiler version as well as the termination code and line function information in the structure *RTNI_lineStatus_s*.

The 'TermType', 'RawPattern', and 'Function' fields in *RTNI_lineStatus_s* contain information to help you handle protocol errors. Providing these fields to technical support greatly reduces the time it takes to diagnose a problem.

TermType

'TermType' contains the terminating code for the last T1 line function. If the function is still running, 'TermType' contains an intermediate message or the protocol code for the previous function. Some protocol codes indicate that the previous function terminated correctly. For example, a protocol code might indicate that the RHT_WAIT_LINE_ON function terminated because it received a call.

Other protocol codes indicate a protocol error. For example, the protocol code might indicate that the RHT_SEND_WINK function terminated because the protocol does not support sending winks. The *RealCT Direct API Reference Manual* contains a complete list of protocol codes and information about how to interpret these values for different functions.

Two common protocol codes are BAD_STATE_ERROR and SIGNALING_BIT_ERROR.

BAD_STATE_ERROR

A BAD_STATE_ERROR indicates that the function could not execute because the line was in the wrong state for that function. For example, 'TermType' would report a bad state error if the application tries to wait for a call or send a wink when the line is not idle. In a bad state error, the CO's interpretation of the previous signal on the line probably didn't match your application's interpretation.

A bad state error could also occur if the application expects a certain event that does not occur, such as a wink after seizing the line in some protocols. Sometimes the signal is not sent because the CO and the application are in different states. This could happen if the application sends a signal that does not meet a minimum time requirement, so the CO does not recognize the signal. The application would be in one state, having sent the signal, but since the CO did not recognize the signal it would not change states. Guard times generally prevent this situation.

The field 'RawPattern' contains the signaling bits being sent and received on the line. This information can help determine why a bad state error occurred, but doesn't explain why the situation arose.

SIGNALING_BIT_ERROR

A signaling bit error occurs when a bit pattern that does not belong in the protocol was present on the line. The invalid bit pattern must be present for longer than the debouncing and deglitching timings in order to generate this error.

If you receive a signaling bit error, compare the bit pattern in *RTNI_lineStatus.RawPattern* to the ones expected by the protocol to be sure your application and the CO are using the same protocol. Call the carrier for more information about their protocol, or use AccuSpan to get more information about the signaling.

If you frequently get a signaling bit error that isn't related to any particular event, you might have a problem with the connection. In this case the application should monitor the status of the carrier. If the number of bipolar violations and buffers slips is high, the line quality is poor. Check the gain your CSU is set to provide, the length of cable, and the equalization parameters set on the hardware.

RawPattern

'RawPattern' contains the received and transmitted signaling bits. The upper byte contains the received pattern while the lower byte contains the transmitted pattern. Each of these bytes contains only four significant bits. These bits contain the value for signaling bits A, B, C, and D. Bit A is the most significant bit, while bit D is the least significant bit.

For example, if 'RawPattern' contains 0x0F05, then the receive pattern is 0x0F and the transmit pattern is 0x05.

Receive=	0x0F=1111, and A=B=C=D=1
Transmit=	0x05=0101, and A=C=0 and B=D=1

Only ESF framing uses all four signaling bits. If your application uses D3/D4 framing, bits C and D are undefined. For example, if 'RawPattern' contains 0x0F05 and you are using D3/D4 framing, then

Receive=	0x0F=1111, and A=B=1
Transmit=	0x05=0101, and A=0 and B=1

and you ignore bits C and D.

Follow the same conventions when using AccuSpan in signaling mode. In D3/D4 framing, bits C and D are not used. If you want to transmit a pattern AB=10, transmit ABCD=10XX =0x08, 0x09, 0x0A, or 0x0B.

In ESF framing, bit A generally equals C and B equals D. So, if you want to transmit a pattern AB=10, set CD=AB and transmit the following:

ABCD=ABAB=1010=0x0A

Function

'Function' shows which function is currently running, or which function terminated last. You need to know the function name in order to understand the terminating code listed in 'TermType'.

The 'Function' field is particularly useful in finding out the cause of BRKT_ERROR_CODE(BUSY). This error means that the line function specified by the 'Function' field was already running when the application called another line function. BRKT_ERROR_CODE(BUSY) is usually a result of a programming error.

Using Loopbacks

Loopback configurations loop data back to where it originated. These configurations help troubleshoot where errors occur. There are several forms of loopbacks, depending on what you want to test.

Set software configurable loopbacks using CONFIG_CARRIER as described in *Configuring the Carrier* on page 52. There are two loopback modes that you specify with CONFIG_CARRIER:

- Test whether the board's receiver and transmitter work properly using a local loopback. In this mode, a trunk receives the same data it transmits without passing through any cabling. To do a local loopback, set the loopback mode in CONFIG_CARRIER to local.
- Test the cabling and verify that the CO can send and receive valid data using a remote loopback. In this mode, the board immediately transmits all data it receives back to the CO. To do a remote loopback, set the loopback mode in CONFIG_CARRIER to remote.

Figure 14 shows local and remote loopbacks.

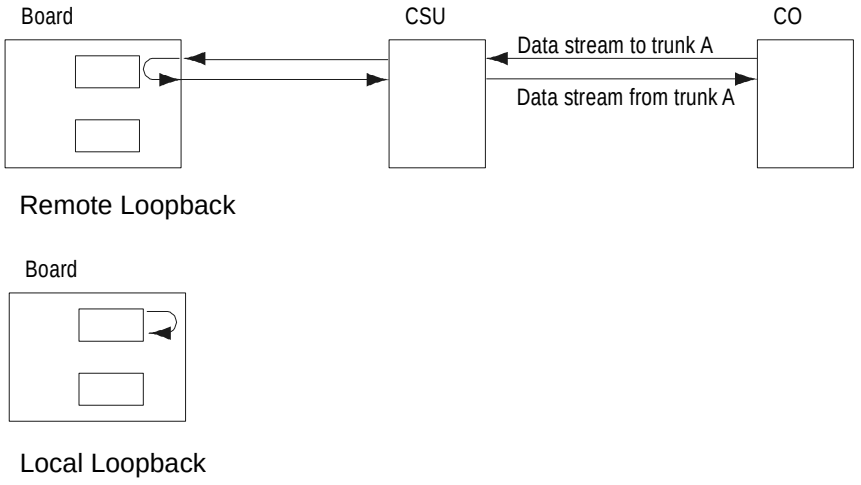

Figure 14. Local and Remote Loopbacks

For other loopback modes, disable both local and remote loopbacks in CONFIG_CARRIER.

- Test the cabling between the CSU and either the board or the CO using a CSU loopback. In this mode, the CSU immediately transmits all data it receives back to the CO or board, depending on which end transmitted the data. To do a CSU loopback, follow instructions from your CSU manufacturer. Figure 15 shows two CSU loopbacks: one looping data back to the CO and the other looping data back to the board.

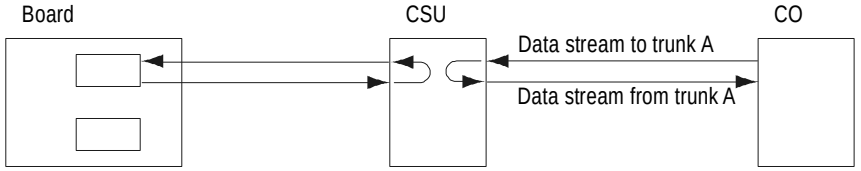

Figure 15. CSU Loopback

- Test individual regions of the cabling using a cable to connect transmit and receive leads at points along a trunk. This mode loops data back to either the board or CO depending on which end transmits the data and how you set up the crossover cable.

  For example, if you have identified that there is a problem with the cabling between the CSU and the board, loop data back at some point along the damaged cable. If the board receives the same data it transmits, the cabling is good between the loopback and the board. The problem lies between the cable and the CSU. Continue testing until you isolate the problem. Figure 16 shows a loopback using a cable to physically connect the transmit and receive leads along trunk A.

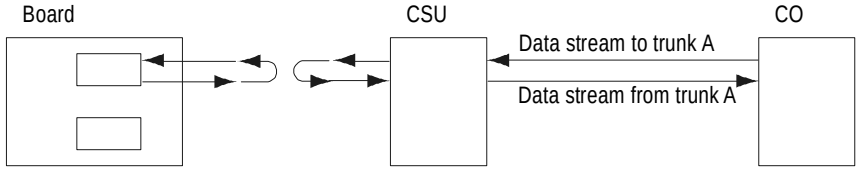

Figure 16. A Cabling Loopback in T1

4
E1 Networking

This chapter describes E1 networking in your system environment.

The E1 board (RTNI-2E1 or NetAccess E1 board) provides two E1 trunks for high-speed communications. The E1 board provides only E1 line resources. It must be in a system with a board that provides voice processing resources over a CT bus such as the Vantage series products.

This chapter includes the following sections:

- Understanding E1 Trunks
- Configuring the E1 Environment
- Handling Incoming and Outgoing Calls
- Using Internal Signaling Streams
- Testing the E1 Setup
- Troubleshooting

Understanding E1 Trunks

E1 trunks provide digital communications in South America, Europe, and Asia. An E1 trunk carries 30 64 Kb/s lines. Two additional 64 Kb/s lines provide signaling information. The combination of the 30 lines and the signaling channels yields a 2.048 Mb/s signal:

32 x 64 Kb/s = 2048 Kb/s or 2.048 Mb/s

The E1 board provides two 30-line E1 trunks for a total of 60 E1 lines. The E1 board only provides E1 line resources; RDSP or Vantage series boards provide voice processing resources through the MVIP bus.

Transmitting Digital Data

Digital data is composed of zeros and ones that the CO and CPE transmit as an electrical signal. The waveform used to indicate zeros and ones is called line coding. The CO and CPE must use the same line coding method in order to communicate properly. As with T1, the E1 boards support Alternate Mark Inversion (AMI) line coding. In AMI, zeros are transmitted as 0V, and ones alternate negative and positive pulses, as Figure 17 shows.

Two subsequent ones with the same polarity are called a bipolar violation. Figure 17 shows a proper AMI signal, with alternating negative and positive ones pulse and two bipolar violations. Bipolar violations could lead to crackling on the line.

To check for bipolar violations, call RHT_GET_STATUS. See *Troubleshooting* on page 134 for more information about how to handle bipolar violations.

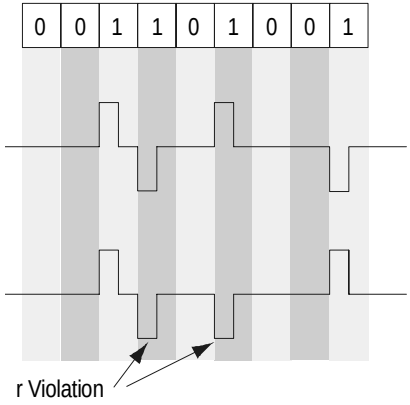

Figure 17. A Bipolar Violation in AMI Coding

In AMI signaling, a long series of zeros is represented by a constant 0V signal. The two ends can lose timing if there is no signal on the line to synchronize them. To maintain ones density on the line, some carriers use a coding method called HDB3 that replace a series of four zeros with a ones-rich pattern. In HDB3 signaling, an inserted one forms a bipolar violation which flags that pattern to be replaced by a series of zeros at the remote end.

E1 boards support line coding with or without HDB3. You set which line coding method your system uses when you configure the carrier as described in *Configuring the Carrier* on page 105.

Organizing the E1 Data

Organizing Data into E1 Frames

The E1 line carries data from 30 data channels plus two channels for framing and signaling over two pairs of wires. At the transmitting end, a multiplexer receives a 64 Kb/s signal from each the 30 lines. It interleaves 8 bits of data from each data channel plus signaling and framing information into a single serial stream of data. The process of interleaving data from each data channel is called time division multiplexing (TDM).

Figure 18 shows data entering a multiplexer and being transmitted on the line. The multiplexer at the far end separates the data into the original 30 data channels plus signaling and framing information.

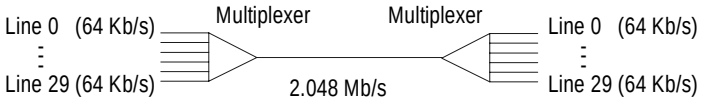

Figure 18. An E1 Line

A multiplexer at the remote end reassembles the multiplexed data into 30 individual data channels plus signaling and framing information.

The transmitting end formats the E1 data into frames so the receiving end can interpret the data. One frame consists of 32 8-bit timeslots. Timeslots 0 and 16 are reserved for framing and signaling information. The remaining 30 timeslots each carry 8 bits of information for a single E1 channel. Timeslots 1RDG_15 carry data for channels 0-14, and timeslots 17-31 carry data for channels 15-29, as shown in Figure 19.

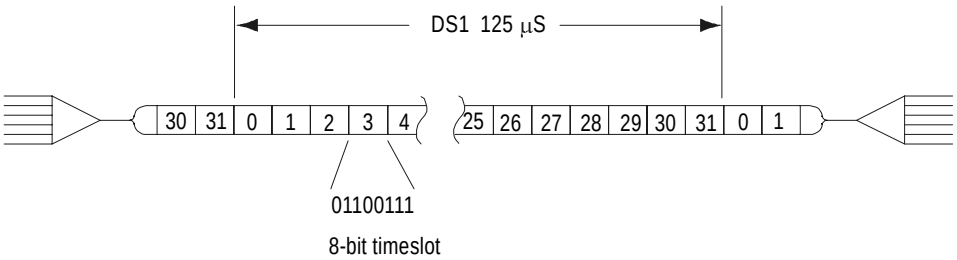

Figure 19. An E1 Frame Carrying 32 8-bit Timeslots

Each frame has a period of 125 µs, or 8000 frames per second. This means that each channel receives 8000 8-bit timeslots per second and operates at 64 Kb/s,

8000 timeslots/second x 8 bits/timeslot = 64 Kb/s

Organizing Frames into CEPT Multiframes

The E1 frames are organized into CEPT multiframes. Each multiframe consists of 16 E1 frames, as Figure 20 shows. Within the multiframes, timeslots 0 and 16 of each frame carry a specific pattern of information.

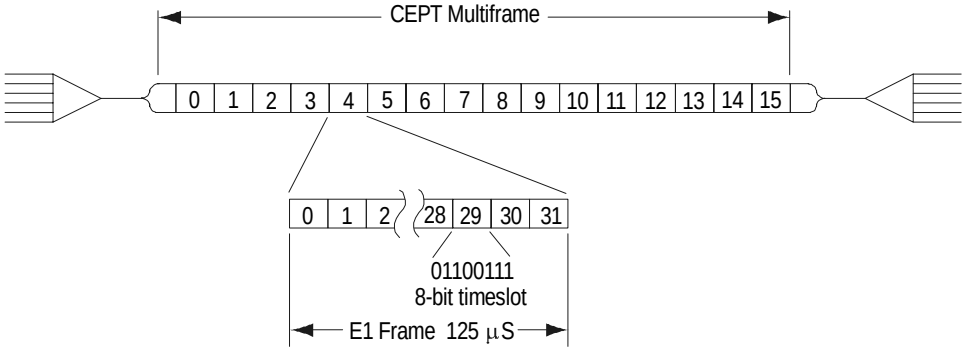

Figure 20. A CEPT Multiframe

Timeslot 0

Timeslot 0 carries framing and CRC information. The bit patterns alternate every other frame.

- In even frames, bits 2-8 follow the framing synchronization pattern: 0011011. The first bit carries cyclic redundancy check (CRC) data.
- In odd frames, the first bit is a CRC alignment signal, the second bit is always 1, the third bit carries alarm information, and the remaining bits are reserved.

Table 17 shows the bit patterns for odd and even timeslot 0 in a CEPT multiframe. The multiframe is divided into two sub-multiframes of eight frames each.

Table 17. Timeslot 0 in a CEPT Multiframe

Frame #			1	2	3	4	5	6	7	8
0	**Framing**	**I**	CRC1	0	0	1	1	0	1	1
1	**Non-Framing**		0	1	A	R	R	R	R	R
2	**Framing**		CRC2	0	0	1	1	0	1	1
3	**Non-Framing**		0	1	A	R	R	R	R	R
4	**Framing**		CRC3	0	0	1	1	0	1	1
5	**Non-Framing**		1	1	A	R	R	R	R	R
6	**Framing**		CRC4	0	0	1	1	0	1	1
7	**Non-Framing**		0	1	A	R	R	R	R	R
8	**Framing**	**II**	CRC1	0	0	1	1	0	1	1
9	**Non-Framing**		1	1	A	R	R	R	R	R
0	**Framing**		CRC2	0	0	1	1	0	1	1
11	**Non-Framing**		1	1	A	R	R	R	R	R
12	**Framing**		CRC3	0	0	1	1	0	1	1
13	**Non-Framing**		Si1	1	A	R	R	R	R	R
14	**Framing**		CRC4	0	0	1	1	0	1	1
15	**Non-Framing**		Si2	1	A	R	R	R	R	R

A=Alarmed

R=Reserved

Framing

The bit pattern in the odd frames synchronizes the transmitting and receiving ends. If a line loses synchronization, the board identifies the framing pattern, then searches for the 1 in the second bit of the next frame. If the board identifies the one, it has regained synchronization.

CRC

The CRC process treats the bits from one sub-multiframe as a single binary number. It performs a calculation and transmits the remainder in the first bit of each odd frame in the following subframe. The receiving end performs the same calculation and compares its results to the transmitted CRC from the remote end. If the two ends calculate the same value, the lines are communicating properly. If the receiving end calculates a different value, the two ends are not communicating properly.

The first bit of the first six even frames contains a fixed alignment signal. The first bit of the last two even frames, called Si1 and Si2, are reserved to transmit the result of the CRC comparison to the remote end. Table 18 shows the meanings for Si1 and Si2.

Table 18. Si1 and Si2 Bits in a CEPT Multiframe

Si1	Si2	Meaning
1	1	CRC results are error free.
1	0	CRC results for SMFII contains errors. CRC results for SMFI are error free.
0	1	CRC results for SMFI contains errors. CRC results for SMFII are error free.
0	0	CRC results for both SMFI and SMFII contain errors.

Timeslot 16

Timeslot 16 of each frame carries the A, B, C, and D signaling bits for each of the 30 data channels. These bits carry information about the line, such as on hook or off hook status. The first four bits of timeslot 16 carry signaling bits for one channel, n, and the second four bits carry signaling bits for channel n + 15. For example, timeslot 16 of frame 1 carries signaling bits for channels 0 and 15 (Timeslots 1 and 17 respectively).

Table 19 shows the frame number and the two channels for which that timeslot carries signaling bits.

Table 19. Timeslot 16 Signaling Bits in a CEPT Multiframe

Frame	Data Channel n	Data Channel n+15
0	Reserved	Reserved
1	0 (TS 1)	15 (TS 17)
2	1 (TS 2)	16 (TS 18)
3	2 (TS 4)	17 (TS 19)
4	3 (TS)	18 (TS 20)
5	4 (TS 5)	19 (TS 21)
6	5 (TS 6)	20 (TS 22)
7	6 (TS 7)	21 (TS 23)
8	7 (TS 8)	22 (TS 24)
9	8 (TS 9)	23 (TS 25)
10	9 (TS 10)	24 (TS 26)
11	10 (TS 11)	25 (TS 27)
12	11 (TS 12)	26 (TS 28)
13	12 (TS 13)	27 (TS 29)
14	13 (TS 14)	28 (TS 30)
15	14 (TS 15)	29 (TS 31)

Configuring the E1 Environment

Configuring the E1 environment involves the following steps:

1. Set the clock.
2. Load the line protocol.
3. Configure the carrier.

Setting the Clock

T1 and E1 boards must have their clock synchronized with the network and with other boards connected in a CT bus. In a CT bus, one board drives the clock and all other boards retrieve the clock from the bus. If the system contains T1 or E1 boards, the first such board drives the clock based on the signal it receives from its network trunk A or B. This synchronizes the CT bus with the T1 or E1 network. Otherwise, a RealBLOCS, RTNI-ATSI, or Vantage PCI board uses its internal oscillator to set the CT bus clock.

Configure the clock using CONFIG_CLOCK or using the Configuration Wizard. For specific information about setting the clock for boards in an CT bus, examples of system configurations, and sample code see *Configuring the MVIP-90 Clock* on page 150.

If you do not set the clock correctly or if you specify more than one board to drive the clock, you could either lose data or transfer data improperly to other boards in the CT bus. An incorrect clock setting can also lead to crackling on the line.

Loading The Line Protocol

The line protocols determine how the central office (CO) communicates with the customer premise equipment (CPE). Both ends must use the same protocol in order to communicate properly. E1 boards support the R2-CCITT standard protocol, and variations for China, Brazil, and Central Europe.

The protocol files convert the API functions such as RHT_ON_HOOK and RHT_OFF_HOOK into the appropriate signaling patterns and timing for the protocol. For example, if an application calls RHT_ON_HOOK, the protocol file handles any signaling or timing involved in going on hook for the current protocol. An application can load a protocol at run time, so one application can use different protocols without modifying any code. Be sure to load the protocol before configuring the line carrier.

Specify the protocol your application uses either when you configure devices with the Configuration Wizard or by calling RHT_LOAD_PROTOCOL. The E1 device uses the R2-CCITT protocol by default. The two E1 trunks on an E1 board can each run different protocols, but all lines on a specific trunk use the same protocol. To find out what protocol file is currently loaded on a given trunk, call RHT_GET_STATUS. This function also provides the file version, source file name, and compiler version used to generate the protocol.

Once loaded, a protocol stays in effect until it is overwritten by a new protocol or the drivers stop. An application should always load the appropriate protocol at runtime to prevent a previously loaded protocol from causing the application to misbehave.

Table 20 shows the protocols that Brooktrout supports and the protocol file names. These files are distributed with the E1 devices. For more information about the E1 line protocols, see Appendix B, *E1 Line Protocols* on page 235.

Table 20. Protocol File Names

Protocol	File name
R2-CCITT	`r2_ccitt.mto`
R2-CCITT (Chinese)	`r2_china.mto`
R2-CCITT (Brazilian)	`r2_brz.mto`
R2-CCITT (Central Europe)	`r2_eur.mto`

The R2-CCITT protocols are symmetrical. The CO and CPE transmit identical bit patterns for each line state. Your application uses the same protocol whether it functions as a CPE or emulates a CO.

Example 5 shows how to load the R2-CCITT protocol using RHT_LOAD_PROTOCOL.

Example 5. RHT_LOAD_PROTOCOL Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{

   BRKT_HANDLE BoardHandle;
   FILE* ProtocolFile;
   BOOLEAN IoctlResult;
   BRKT_SIZE_T BytesReturned;
   BRKT_SIZE_T FileSize, BytesRead;
   USHORT* pProtocolBuffer;
   struct LoadProtocol_s Protocol;

   BoardHandle = BrktOpenDevice (BRKT_DEVICE_E1_BOARD, 0);

   /* Open Protocol File */
   ProtocolFile = fopen ("R2_CCITT.MTO", "rb");

   FileSize = _filelength (_fileno (ProtocolFile);
   pProtocolBuffer = malloc (FileSize);

   if (pProtocolBuffer == NULL)
   {
         printf ("Can't allocate memory for protocol.\n");
         return(2);
   }

   BytesRead = fread (pProtocolBuffer, sizeof (UCHAR), FileSize,
                      ProtocolFile);
   fclose (ProtocolFile);
```

Example 5. RHT_LOAD_PROTOCOL Sample Code (Continued)

```
   memset (&Protocol, 0, sizeof (Protocol));
   Protocol.Length = FileSize;
   Protocol.Protocol = pProtocolBuffer;
   Protocol.Trunk = M_TRUNK0;      /* Trunk 0 */

   IoctlResult = BrktDeviceIoControl (
         BoardHandle,
         RHT_LOAD_PROTOCOL,
         &Protocol,                /* Buffer to driver */
         sizeof(Protocol),         /* Length */
         NULL,                     /* Buffer from driver */
         0,                        /* Length */
         &BytesReturned,
         NULL);                    /* Wait till I/O complete */

   if (!IoctlResult)
         printf ("LOADPR failed: BrktGetLastError = %d\n",
               BrktGetLastError (BoardHandle));
   else
         printf ("LOADPR done \n");
   BrktCloseDevice(BoardHandle);
   return(0);
}
```

Configuring the Carrier

Configure the carrier parameters for your line using the function CONFIG_CARRIER, or configure only line coding and CRC using the Configuration Wizard. Using CONFIG_CARRIER, you configure all channels on a given trunk to use the same values. To configure the ADI, invert, or loopback parameters separately for individual channels, use CONFIG_CHANNEL and RHT_CONFIG_CHANNEL. However, since there is rarely a need to configure these parameters separately, we recommend using CONFIG_CARRIER.

For all parameters, contact your carrier for information about the appropriate settings.

Configuring CRC

You can enable or disable your system to transmit CRC data and the CRC error report. The CRC calculation and error report provide a good way to verify the data transmitted on the line and to analyze the data link in both directions.

Setting the CRC parameter incorrectly causes the board to indicate framing errors, which causes line functions to fail.

Configuring the Line Coding Method

E1 boards support line coding with or without HDB3. Setting the incorrect line coding method can cause bipolar violations or noise on the line, so be sure your configuration matches the specifications of your carrier.

Configuring Debounce

When debouncing is enabled, the board waits for the signaling bits to be stable before relaying the information to the devices. Waiting for the bits to stabilize prevents errors from being handled as if they were valid signals. Debouncing, which is also called deglitching, should always be enabled.

The ‘6’ to ‘9’ msec wait is less than the minimum time used for signal recognition protocols, so enabling debouncing should not impact performance. In fact, it should increase performance, despite the delay, since the software does not have to handle spurious signals. The software adds an additional debouncing time at a higher layer, which further reduces the risk of spurious signals reaching the application.

Configuring the Loopback Mode

There are two software loopbacks set using CONFIG_CARRIER: remote and local. These are used to test a board or the data link. For information about additional hardware loopbacks, see *Troubleshooting* on page 134.

- In a remote loopback, the board immediately transmits all data it receives back to the CO. If all connections are working, the CO should receive the same information it transmitted. If any transmission errors disappear in the remote loopback configuration, the problem is with the clock or carrier settings in your application. If the problem does not disappear, the problem is either the cabling or at the carrier end.
- In a local loopback, the board receives the same data it transmits, without passing through any cabling. If placing your board in a local loopback fixes problems you experienced on the line, the problem is with the cabling or with the carrier. If a local loopback does not fix errors, then the board is not sending valid data.

When you first load the driver, it has both loopback modes enabled by default. In this configuration, both the board and the CO only receive the data they transmit. This configuration lets you verify that both the board and CO can send and receive valid data. For normal operation, remove both loopbacks so that the board and CO can transmit and receive data.

Configuring the Hook State

When loaded, each protocol automatically transmits the bit pattern corresponding to the idle state. The devices ignore the 'Hook' field in the carrier parameters.

Transmitting the proper hook state is one reason to load the protocols before configuring the carrier parameters. If you configure the carrier first, the board might transmit an invalid hook pattern on the line. This invalid signal could trigger alarms at the CO. Although the alarms would disappear when you load the line protocol and begin transmitting the appropriate bit pattern, the CO switch might have blocked the line in response to the alarm.

Configuring ADI

Alternate Digit Inversion (ADI) consists of analyzing the data transmitted in each channel and making sure it does not go beyond certain amplitude limit. If it does, the signal is automatically modified so that it remains inside a pre-determined range. For audio data, ADI should always be enabled.

Example 6. CONFIG_CARRIER Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{

   BRKT_HANDLE BoardHandle;        /* E1 board device handle */
   BOOLEAN IoctlResult;            /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;      /* Bytes returned from IOCTL call*/
   struct RTNI_E1carrierParam_s Carrier;

     /* Open T1 board device */
   BoardHandle = BrktOpenDevice (BRKT_DEVICE_E1_BOARD, 0);

     /* Configure E1 carrier */
     /* Set up same parameters for both trunks */
   memset (&Carrier, 0, sizeof (Carrier));
   Carrier.Size = sizeof(struct RTNI_E1carrierParam_s;
   Carrier.Trunk = M_ALL_TRUNK;
   Carrier.CRC = 1;                /* Enable CRC calculation */
   Carrier.Code = DT_HDB3;         /* Coding method: HDB3 encoding */
   Carrier.Debounce = 1;           /* Enable debounce (deglitch) */
   Carrier.CCS = 0;                /* No CCS. Use CAS signaling */
   Carrier.Loopback = 0;           /* Disable loopback */
   Carrier.Alarm = 0;              /* Do not send alarms */
   Carrier.Hook = H_ON;            /* On-hook (idle) state */
   Carrier.ADI = TRUE;             /* Enable ADI encoding */
   Carrier.TxGain = 0;             /* Sets trunk’s transmit gain at 0 dB */
   Carrier.RxGain = 0;             /* Sets trunk’s receive gain at 0 dB */
```

Example 6. CONFIG_CARRIER Sample Code (Continued)

```
   IoctlResult = BrktDeviceIoControl (
         BoardHandle,
         CONFIG_CARRIER,
         &Carrier,                /* Buffer to driver */
         sizeof(Carrier),
         NULL,
         0,
         &BytesReturned,
         NULL);                   /* Wait until I/O is complete */
   if (!IoctlResult)
         printf ("CONFIG_CARRIER failed: BrktGetLastError = %d\n",
               BrktGetLastError (BoardHandle));


   BrktCloseDevice(BoardHandle);
   return(0);
}
```

Handling Incoming and Outgoing Calls

Processing Calls

Call processing involves setting up and tearing down calls. Calls have to be processed before they go through, so be sure to verify call processing in the early stages of development.

Table 21 shows the functions used in sending and receiving calls over an E1 line. These functions are the same as those used in analog telephony. The protocol files translate these functions into the specific bit patterns and handshaking signals used by the E1 protocols, as described in *Loading The Line Protocol* on page 101.

Table 21. API Functions Used in E1 Signaling.

Function	Description
RHT_WAIT_LINE_ON	Waits for an incoming call and automatically acknowledges it.
RHT_OFF_HOOK RHT_SEIZE_LINE	Answers and incoming call or initiates an outgoing call.
RHT_ON_HOOK RHT_DISCONNECT	Terminates incoming or outgoing calls.
RHT_STOP	Terminates a function.
RHT_BLOCK_LINE	Blocks the line, preventing the line from receiving incoming calls.
RHT_FORCED_RELEASE	Terminates an incoming call by transmitting a Forced Release signal on the line.
RHT_WAIT_LINE_OFF	Monitors the line waiting for a disconnect
RHT_WAIT_ANSWER	Monitors the line waiting for an answer.
RHT_WAIT_IDLE	Waits for an idle pattern on the line.

Handling Incoming Calls

Handling incoming calls involves the following steps:

1. Detect an incoming call.
2. Handle call setup signaling.
3. Send control tones.
4. Answer the call.
5. Process the call, while monitoring for disconnect.
6. Terminate the call.

You can block a channel at any time to prevent further incoming calls.

Detecting an Incoming Call

When the application is ready to receive a call, it calls RHT_WAIT_LINE_ON. This function waits until the device receives a seizure signal, then sends a seizure acknowledgment. The function waits for an amount of time specified by the *RDG_LOCAL_ACK_GUARD_TIME* parameter, then returns an acknowledgment signal. The application is then ready to receive digits. The series of events involved in detecting an incoming call is shown in Figure 22.

Table 22. RHT_WAIT_LINE_ON Sequence

Action	Duration
Receive seizure or ring	
Send acknowledgment (if required)	
Wait	*RDG_LOCAL_ACK_GUARD_TIME*
Send acknowledgment	
Return	

RHT_WAIT_LINE_ON automatically sends any acknowledgments required by the protocol, which serves two purposes:

- First, it isolates the application from the protocol. You can communicate with different carriers simply by switching protocol files rather than making changes to the application.
- Second, it improves timing. In a heavily loaded system, threads and processes can be preempted at almost any time, and it might take several seconds before they can run again. If there were separate functions to detect the call and send the acknowledgment, the two signals could be several seconds apart. This delay might exceed the maximum allowed by the originating end.

If the application calls RHT_WAIT_LINE_ON when the line is in a state other than idle, the function returns an error.

RHT_WAIT_LINE_ON should continuously wait for a call, even though you can limit the function's run time with the *MDP_WAIT_LINE_ON_TIMEOUT* parameter. If you use this timeout to monitor for other conditions, you could miss incoming calls. Use another thread to do any necessary monitoring. To terminate RHT_WAIT_LINE_ON, use RHT_STOP. The application is notified that the function terminated and will take any appropriate action.

Detecting Digits

When an application receives an incoming call, it generally receives information about the call setup from the originating end. The method used to send these digits depends on the way the CO has configured the line. Most COs use R2 compelled signaling, but some use DTMF or MF tones to send digits.

Despite the name, there is no association between the R2-CCITT line signaling protocols and the R2 inter-register signaling. Any board with voice processing capabilities can handle R2 inter-register signaling, regardless of the type of line interface. Likewise, an E1 line configured to use an R2-CCITT line protocol can receive digits through a different method of inter-register signaling. However, most E1 systems use R2 inter-register signaling. For more information about R2 inter-register signaling, see Chapter 2, *Digit Handling,* on page 15.

Before detecting new digits from an incoming call, flush the digit buffer using RHT_FLUSH_DIGIT. Otherwise, digits stored in the buffer from a previous call could be handled in the new call. The best time to flush digits is after going on hook but before calling RHT_WAIT_LINE_ON. If you flush digits after calling RHT_WAIT_LINE_ON, your thread could get pre-empted and rescheduled for after the application has started receiving digits from a new call. When the application eventually flushes the digit buffer it clears digits from the current call and loses that data.

Once the application receives call setup information, it answers the call. If the inter-register signaling fails, then the circuit is released.

Sending Control Tones

In R2-CCITT protocols, your application emulates both the receiving end CO and subscriber. The CO portion sends control tones such as ringback and busy to indicate the status of the call to the other end.

When a subscriber receives a call, the CO emulation part of the application should determine the status of the line and send either a busy or ringback tone back to the caller. It should also send the hangup tone when the subscriber hangs up. Many applications skip sending control tones and instead send the answer signal immediately after receiving the incoming call, leaving the line silent until the application is ready to start playing prompts or a live person can handle the call. If a calling party does not hear control tones, they might think the call was lost and hang up, especially if the application had to perform a lengthy task such as querying a data base or checking the status of agents at a call center.

Sending control tones also improves the timing of the call if there are many transit COs between the calling party and your application. In this case, it could take several hundred milliseconds to establish an audio path between the two ends. If the application immediately begins playing audio prompts after receiving a call, the caller could lose the first portion of the prompt that was played while the audio path was being established. Sending control tones establishes the audio path before the application begins playing, so the caller hears the entire audio prompt.

Although adding the control tones might add some complexity, some countries require that your application meets some minimum requirements before live traffic is routed to it.

Answering Calls

Answer calls detected by RHT_WAIT_LINE_ON using RHT_OFF_HOOK or RHT_SEIZE_LINE. These functions are interchangeable, so all discussion of RHT_OFF_HOOK also applies to RHT_SEIZE_LINE.

When RHT_OFF_HOOK answers a call, it automatically transmits the appropriate answer supervision signal towards the switch directly connected to your system. The switch relays this information back to the preceding switching equipment and so on until the information reaches the originating equipment. This serves as a signal to the originating office to start billing the caller.

After transmitting the off hook or answer supervision bit pattern, RHT_OFF_HOOK waits for a duration specified by *RDG_LOCAL_ANSWER_GUARD_TIME* before returning, as shown in Table 23.

Table 23. RHT_OFF_HOOK Sequence

Action	Duration
Transmit answer signal	
Wait	*RDG_LOCAL_ANSWER_GUARD_TIME*
Return	

The guard time is important in case your application needs to hang up immediately after answering a call. Without a guard time, the off hook pattern would be on the line for a very short period of time. The CO could ignore that signal, or consider the transmission to be an error. Although this situation is correctly handled in most protocols, it could be a problem in some specific scenarios.

In the early stages of development, it might be hard to tell if the application fails to call RHT_OFF_HOOK. The audio path is established soon after the CO sends digits so the remote end can hear the control tones. With the audio path established, the remote end will also hear any files the application plays even if the application fails to call RHT_OFF_HOOK. If the CO does not receive the answer signal, however, it times out within two to four minutes and terminates the call. To the application and the caller, it appears that the call was being handled properly until the call is suddenly terminated. If your application consistently reports that it received a disconnect after two to four minutes while it was processing the call, check to be sure the application called RHT_OFF_HOOK to answer the call.

If you are sure that your application called RHT_OFF_HOOK, consider the other extreme. The application might call RHT_OFF_HOOK too soon after it detects digits. In this case the switches down the path might not be ready to receive the answer signal, so they do not detect it. The application should send control tones after receiving digits so the COs in the path are ready to receive the answer signal.

Monitoring For Disconnection

Whether you received or originated the call, you should continuously monitor the line for disconnect using RHT_WAIT_LINE_OFF. The sooner you detect the disconnect, the sooner the line is free for new calls, allowing you to reach more people in the same amount of time without increasing the number of lines. RHT_WAIT_LINE_OFF returns successfully when it detects a disconnect or returns an error if it detects other conditions. If RHT_WAIT_LINE_ON returns an error, use RHT_GET_STATUS for more information about the termination.

If the application is running a voice processing function when it detects disconnect, it should terminate that function immediately. There are two ways to terminate a function. The first uses two threads (or processes) per channel: one to run the line monitoring function and another the run the voice processing function. The main thread synchronizes these two threads.

A better way to terminate VP functions is to run the functions so they terminate when the application detects a disconnect. To do this, set the field 'LineTerm0' in the *VPstartStop_s* or *RhtDialDigit_s* structures. This field provides a way for the VP driver and the E1 device to communicate.

With 'LineTerm0' set, the E1 device automatically runs RHT_WAIT_LINE_OFF. It continues monitoring the line until the VP driver sends a signal that the VP function terminated or until the E1 device detects a disconnect. When it detects a disconnect, the E1 device signals the VP driver to terminate the function. When the VP function terminates, check the condition that caused termination using RHT_GET_STATUS. A T_RHT_LINE_OFF condition means that RHT_WAIT_LINE_OFF caused the function to terminate.

This second approach is easier to implement because it does not involve separate threads. However, RHT_WAIT_LINE_OFF only runs when a VP function is running. If the application spends long periods of time performing non-VP functions, a disconnect would not be reported until the next VP or line function runs. This could keep the line busy longer than necessary. If this delay is not acceptable, then use a separate thread to monitor the line while no VP functions are running.

Since RHT_WAIT_LINE_OFF is an exclusive function, you cannot start any other exclusive functions when you start a VP function with 'LineTerm0' set.

Terminating an Inbound Call

Terminate calls using RHT_ON_HOOK or RHT_DISCONNECT. These functions are interchangeable, so all discussion of RHT_DISCONNECT also applies to RHT_ON_HOOK.

RHT_DISCONNECT transmits a disconnect (idle) pattern for the time specified by *RDG_LOCAL_IDLE_DUR*. It then waits for the remote end to disconnect for a time specified by *RDG_REMOTE_IDLE_TIMEOUT*. After it receives the disconnect, RHT_DISCONNECT waits for a time specified by *RDG_LOCAL_IDLE_GUARD_TIME* then returns. Table 24 shows the sequence involved in terminating a call.

Table 24. RHT_DISCONNECT Sequence

Action	Duration
Transmit disconnect	*RDG_LOCAL_IDLE_DUR*
Wait for disconnect	*RDG_REMOTE_IDLE_TIMEOUT*
Wait	*RDG_LOCAL_IDLE_GUARD_TIME*
Return	

In the R2 CCITT protocol, the protocol sends a Clear Back disconnect signal to terminate incoming calls. Outgoing calls terminate using a Clear Forward signal.

If the remote end disconnects first and your application calls RHT_DISCONNECT in response to their disconnect signal, it returns almost immediately. The only delays are those caused by *RDG_LOCAL_IDLE_DUR* and *RDG_LOCAL_IDLE_GUARD_TIME*.

The remote end only sends an idle pattern when the calling party hangs up. If the caller does not know that your application has disconnected, it might take a while for them to hang up. An application should play a busy signal when it hangs up so the calling party knows you have disconnected.

If your end disconnects first, RHT_DISCONNECT waits for a duration specified by *RDG_REMOTE_IDLE_TIMEOUT* for the other party to disconnect. This parameter is set to infinite by default. However, the CO the other party is connected to usually times out in one to four minutes if the other party does not disconnect. When it times out, the CO transmits a disconnect signal on its own. If the CO does not have a timer, it does not transmit a disconnect and your application must wait for the other party to hang up.

If RHT_DISCONNECT does not return within a few seconds, it is probably because the other party has not hung up rather than a problem with the application. Call RHT_GET_STATUS to see the signaling bits currently present on the line if you think the function is taking too long to return.

The reason RHT_DISCONNECT waits for a disconnect before returning is so the application knows when the line is free. If RHT_DISCONNECT returned immediately, the application would not know when the line could be used for new calls. The application would have to call RHT_WAIT_IDLE to monitor the line for disconnect after RHT_DISCONNECT returns.

You can change *RDG_REMOTE_IDLE_TIMEOUT* to be less than infinite, but doing this does not free the line any faster. If RHT_DISCONNECT does not receive the disconnect signal within a time specified by *RDG_REMOTE_IDLE_TIMEOUT*, the function returns an error and the application resumes executing. However, the application cannot make or receive another call until the line is idle, so you do not gain anything by regaining control. Any exclusive line functions also return an error until the line becomes idle. The only way the application knows when the line becomes free is by calling RHT_WAIT_IDLE. This function returns when the line becomes idle and the application can proceed. Since RHT_WAIT_IDLE is built into RHT_DISCONNECT, it is most efficient to wait for RHT_DISCONNECT to detect the idle and return.

If you are using R2 inter-register signaling and you use RHT_DISCONNECT to terminate a call, the protocol sends a Clear Back signal to disconnect. This signal uses the same bit pattern as the Seizure Acknowledgment signal (A=1, B=1). The far end can not distinguish between the two signals, so the Clear Back has no effect if the application called RHT_DISCONNECT from the seizure acknowledgment state. The far end continues inter-register signaling, which eventually fails because the near end does not respond. When inter-register signaling fails, the far end terminates the call and releases the line for future calls.

Some countries extend the CCITT R2 protocol to support the Forced Release signal, which provides an alternate way to terminate inbound calls. The Forced Release signal has a bit pattern that is different from other backward signals, so it is easily recognizable to the remote end. Use the RHT_FORCED_RELEASE function to transmit the Forced Release signal. This function returns an error if is called to terminate an outbound call.

The Forced Release signal is not part of the CCITT recommendations, and is not always supported. Check with your carrier to find out it they support Forced Release.

Blocking a Circuit

It might be necessary for an application to stop receiving calls, for example if the system is down for maintenance. In an analog environment, the system can simply ignore incoming calls. In an E1 environment using R2 line signaling, however, ignoring calls could lead to problems with the CO.

When the far end sends a Seizure signal, the application responds with a Seizure Acknowledgment. If the application cannot accept calls and does not send a Seizure Acknowledgment, the CO might consider the circuit to be faulty and take the line out of service.

To prevent the CO from taking lines out of service, Block the line using RHT_BLOCK_LINE. The Block signal also acts as a Clear Back signal, terminating any incoming calls. Check with your carrier to find out how the Block signal handles outgoing calls. In some systems you need to terminate outgoing calls before blocking the line. Once the application is ready to start processing calls, call RHT_ON_HOOK to transmit an Idle pattern on the line.

If your application tries to make an outbound call and detects a blocked line, the line function returns error BRKT_ERROR_CODE(IO_DEVICE). The application calls RHT_WAIT_IDLE to detect when the circuit is Idle and can receive calls.

Handling Outbound Calls

Handling outbound calls involves the following steps:

1. Seize a line.
2. Send call-setup information (R2 Inter-register Signaling or other proprietary methods).
3. Monitor the line for answer.
4. Process the call.
5. Terminate the call.

Seizing a Line

Seize an idle line using either RHT_SEIZE_LINE or RHT_OFF_HOOK. These functions are interchangeable, so all discussion of RHT_SEIZE_LINE also applies to RHT_OFF_HOOK.

RHT_SEIZE_LINE transmits the line seize signal and waits for an acknowledgment from the remote end for a time defined by *RDG_REMOTE_ACK_TIMEOUT*. It then waits for a time determined by *RDG_LOCAL_SEIZE_GUARD_TIME* and returns. If RHT_SEIZE_LINE does not receive an acknowledgment, it returns the hook to idle and returns an error. Table 25 shows the sequence of seizing a line.

Table 25. RHT_SEIZE_LINE Sequence

Action	Duration
Transmit line seize	
Wait for acknowledgment	*RDG_REMOTE_ACK_TIMEOUT*
Wait	*RDG_LOCAL_SEIZE_GUARD_TIME*
Return	

Glare Resolution

If the line is not idle when the application calls RHT_SEIZE_LINE, BrktGetLastError (DeviceHandle), returns BRKT_ERROR_CODE(IO_DEVICE). See *Troubleshooting* on page 134 for more information on how to handle the error.

It is possible for the application to try to seize the line as the CO tries to send a call. In this situation, called a glare, RHT_SEIZE_LINE returns an error but does not set the line back to idle. In order to accept the incoming call, the application calls RHT_OFF_HOOK. If it does not accept the call, it calls RHT_DISCONNECT to set the line back to idle. Check with your carrier to see how they want applications to handle glare situations (also called glare resolution).

Sending Call Setup Information

After the application seizes the line, it transmits call setup information using RHT_DIAL_R2. Call setup is usually done using R2-MF tones, also called MFC. RHT_DIAL_R2 handles all aspects of the compelled protocol.

While dialing, the application monitors the line by setting field 'LineTerm0' in the *VPdialR2_s* structure as described in *Monitoring For Disconnection* on page 117.

If the function returns successfully, the application monitors the call for answer. If the function returns T_RHT_LINEOFF, then the function terminated because of a line condition. Call RHT_GET_STATUS to find out what line condition occurred. Other codes indication more serious error conditions. You should abandon the call using RHT_DISCONNECT and try the call again.

For more information about how E1 applications dial digits, see Chapter 2, *Digit Handling,* on page 15.

Monitoring For Call Answer

Determine the status of a call using either RHT_START_PCPM or RHT_WAIT_ANSWER. RHT_WAIT_ANSWER returns successfully if it detects an answer pattern or returns an error if it detects any other pattern such as a disconnect or bit errors. If RHT_WAIT_ANSWER does not detect a bit change, it continues running until the application terminates it through a call to RHT_STOP.

In some protocols, the bit patterns are the same when dialing out and when the remote end sends a disconnect signal. To distinguish between these, call RHT_WAIT_ANSWER immediately after dialing out and before calling any other line function. After RHT_WAIT_ANSWER detects the answer, call RHT_WAIT_LINE_OFF to detect disconnect. Calling the functions in this specific order allows the device to keep track of the call history and to differentiate between ambiguous bit patterns based on their context.

RHT_WAIT_ANSWER only monitors for an answer based on the answer supervision signal. If the application needs more information about the call than whether it was answered, use RHT_START_PCPM. RHT_START_PCPM accesses the Programmable Call Progress Monitoring (PCPM) algorithm that runs on the board's DSP. PCPM monitors the line for control tones, voice, or silence on the line and determines the status of the call. Since PCPM monitors all audio on the line, it can provide information about busy, no answer, or other line conditions.

In systems using R2 inter-register signaling, an R2 signal indicates whether the line is busy. Also, a bit change in the line signaling protocol reports when the line becomes free, so PCPM is not necessary to detect busy. For more information about R2 signaling, see Chapter 2, *Digit Handling,* on page 15.

The main reason to run PCPM algorithms in R2 systems would be detecting special tones such as Fax tones or operator intercept not accompanied by an R2 register signal. However, proprietary protocols running on E1 lines might require PCPM. RHT_START_PCPM only monitors audible signals on the line and runs independently of the type of line being used.

Since PCPM uses only sound or silence to determine the call status, it cannot be absolutely accurate in detecting answer. For example, if a busy switch plays a recording saying that all circuits are busy, the PCPM algorithm might recognize the human voice and determine that the call has been answered. For this reason, you should run RHT_WAIT_ANSWER and RHT_START_PCPM simultaneously. RHT_WAIT_ANSWER provides accurate answer detection while RHT_START_PCPM provides other information about the status of the call.

If the application starts RHT_START_PCPM with *VPstartStop_s.LineTerm0* set, the VP driver automatically requests that the E1 device run RHT_WAIT_LINE_OFF. RHT_WAIT_LINE_OFF monitors the line for a disconnect, but it also detects answer. When RHT_WAIT_LINE_OFF detects an answer or disconnect, it terminates RHT_START_PCPM. The application then calls RHT_GET_STATUS to determine the status of the line.

The following example shows the steps involved in using RHT_START_PCPM to detect answer.

1. The application starts RHT_START_PCPM with *VPstartStop_s* fields 'LineTerm0' and 'Timeout' set.
2. When RHT_START_PCPM returns, the application calls RHT_GET_STATUS for the VP device and checks *VPchanStatus_s.TermType*.

Interpreting Results

1. If *VPchanStatus_s* = T_RHT_PCPM, then RHT_START_PCPM determined the line state. The *VPchanStatus_s.PCPMtype* field contains the line status information.
2. If *VPchanStatus_s.TermType* = T_RHT_TIMEOUT, then RHT_START_PCPM did not detect any condition within the allotted amount of time.
3. If *VPchanStatus_s.TermType* = T_RHT_LINEOFF, the CO either answered or disconnected the call before answering. The application calls RHT_GET_STATUS for the line device and checks *RTNI_lineStatus_s.TermType*. This field contains information about the status of the line.

Terminating an Outbound Call

Terminating an outbound uses the same procedure as terminating an inbound call, except that outbound calls terminate using a Clear Forward rather than a Clear Back signal. RHT_DISCONNECT handles this distinction. For more information about terminating calls, see *Terminating an Inbound Call* on page 119.

Using Internal Signaling Streams

The E1 board has internal streams that hold data received on the line. Streams 17 and 19 hold signaling bits from trunks A and B, respectively. Streams 16 and 18 hold data from trunks A and B, respectively.

Within streams 17 and 19, 8-bit timeslots hold the signaling data. The internal MVIP timeslots hold signaling bits the same way they are transmitted in the E1 timeslot 16: the first nibble holds signaling bits for line n and the second nibble holds signaling bits for line n+15.

For example, timeslot 0 in stream 17 holds signaling bits for line 0 and line 15. Timeslot 1 holds signaling bits for line 1 and line 16, and so on through timeslot 14, which holds signaling bits for lines 14 and 29. If timeslot 0 transmits the value 0x9D, then line 0 is transmitting ABCD=0x9=1001, and line 15 is transmitting ABCD=0xD=1101.

Access the signaling bits using the SET_OUTPUT and SAMPLE_INPUT MVIP functions. Applications can use these functions to set and retrieve signaling bits directly from the hardware instead of relying on higher level device functions such as RHT_GET_LINE and RHT_GET_STATUS. Access to low level signaling bits should only be used for tracing purposes, since it can interfere with normal device functions such as RHT_ON_HOOK.

To write bits to a specific internal stream/timeslot, put the timeslot in message mode and use SET_OUTPUT. The hardware takes the bits you write to the signaling streams and sends them as signaling bits. To read bits from a specific stream/timeslot, use SAMPLE_INPUT. See the *RealCT Direct API Reference Manual* for more information about SET_OUTPUT and SAMPLE_INPUT.

The E1 board receives data for lines 0 through 29 in internal streams 16 and 18, timeslots 0 through 29. For information about switching data received in streams 16 and 18 over the CT bus, see Chapter 5, *MVIP-90,* on page 143, and Chapter 6, *MVIP-95,* on page 183.

Testing the E1 Setup

Testing the Installation

Use the samples that came with your software to test your boards, devices, and installation before you write your application. You can use the provided source code to help develop your application. The samples are located in subdirectory \RHT\SAMP\E1.

The most useful samples are:

E1INIT	Sets the clock, initializes carrier parameters, and checks carrier status. Older samples set the clock reference as being Trunk A and configured the carrier to disable HDB3 and CRC. If your installation requires different values, make sure to alter the samples accordingly. Newer samples configure most of the parameters using the command line. The defaults are the same as those in the older samples.
WAITRING [line]	Waits for a call on the specified line (0 by default).
OFFHOOK [line]	Answers an inbound call or initiates an outbound call.
ONHOOK [line]	Disconnects a call on the specified line (0 by default).
DISLN [line]	Disconnects a call on the specified line (0 by default).
WAITANS [line]	Waits for the remote end to answer on the specified line (0 by default).
BLKLN [line]	Blocks the line.
FRLN [line]	Sends a Forced Release signal to terminate the call.
WAITOFF [line]	Monitors for a disconnect on the specified line (0 by default).
WAITIDLE [line]	Waits for an idle pattern on the line (0 by default).
LSB [line]	Displays all relevant information about a line (structure *RTNI_lineStatus_s*).
PLAYE [line] [file]	Plays a file while monitoring for a disconnect on the line (default: line 0, file test.vox).

DIALR2 [line] [digits]	Dials R2 digits while monitoring for a disconnect on the line (0 by default).
BSTAT	Queries carrier status.
BINFO	Displays board and device information.
CONN	Makes MVIP connections. Usage: `Conn<output stream> <output timeslot> <input stream> <input timeslot>`
QUERY	Queries MVIP connections. Usage: `Query<output stream> <output timeslot>`
DIGIT [line]	In subdirectory \RHT\SAMP\STD. Reads digits sent by the CO.
TWAIT [line]	In subdirectory \RHT\SAMP\STD. Waits for digits. It must be modified so that it monitors an E1 line for disconnect instead of a LS line.
SETDM [line]	In subdirectory \RHT\SAMP\STD. Selects the type of digits to detect. Call this to enable R2 forward or R2 backward digits before you call DIALR2.

Run these samples either individually or as a batch file. For example, the following batch file initializes the E1 line, makes the MVIP connection between line 0 and VP0, receives an inbound call, plays a file, and then disconnects:

```
E1INIT
CONN 16 0 6 1
CONN 6 1 16 0
WAITRING 0
TWAIT 0
DIALR2 0
      Repeat the above two steps until signaling is
      complete
OFFHOOK 0
PLAYE 0 TEST.VOX
ONHOOK 0
```

In this example, TWAIT has to be modified to monitor an E1 line rather than an LS line. Modify DIALR2 to handle multiple digits instead of just one.

Testing the Application

There are four basic ways of testing your application:

- Using AccuSpan
- Placing calls over a live E1 line
- Using an E1 simulator
- Connecting trunks A and B (Loopback)

Using AccuSpan

AccuSpan is the best way to test the overall robustness and fault tolerance of your system. AccuSpan is a command line utility that gives you full control over the carrier configuration, alarms, and bit patterns sent and received. It can also receive or generate calls. By changing the carrier parameters such as clock settings, line coding methods, or CRC transmission, you can test different scenarios for your application. You can also transmit alarm conditions to see how the remote end responds. To use AccuSpan, connect two back to back computers with one running the application and the other running AccuSpan.

The most powerful feature that AccuSpan provides for exception-handling tests is signaling mode. In signaling mode you can transmit any possible bit pattern to the application, where the received bit pattern is always shown on screen. By sending certain bit patterns at specific times you can manually generate all events the protocol supports, such as line seizure, answer, or disconnect. You can than send valid events at invalid times, send invalid bit patterns at several different states of the call progress, or send signals that are too long or too short and check how the application responds. Using signaling mode requires a good understanding of the protocols. If you are not already familiar with the protocols, view the bit patterns sent or received on the line during normal operation to see how the protocol operates.

Placing Calls over an E1 Line

Testing your application over a live E1 line gives you a controlled environment and requires no special knowledge of the protocols. If the application passes a few tests such as making simultaneous calls or consecutive calls on the same line, the application will probably work for real calls. However, when testing the application over a real E1 line, the telephone company controls the environment so there is little chance to test error conditions. Before testing an application on a live E1 line you should use AccuSpan to eliminate as many error conditions as possible.

Using an E1 Bulk Call Generator

Using an E1 bulk call generator is a good way to stress-test your application. Most E1 bulk call generators allow at least four trunks, or 120 lines, to place calls simultaneously. However, using a call generator should not replace other tests, particularly tests over a real line. The bulk call generator usually does not provide a way of testing error handling since they follow scripts that you determine and send the correct signals on the line. They might also handle exceptions and timing issues differently than your CO.

Connecting Trunks A and B

When you connect trunks A and B, you receive data on one trunk that you send from the other trunk. Using this configuration, you can test samples that compliment your application or use the samples that came with your software. Using either your own or provided samples, you can test inbound and outbound call processing, send events out of order to see how the application reacts, and stress test the application. A loopback configuration is normally used as an initial test, but it cannot test line handling features of the application.

When you connect both trunks, be sure that the transmit leads from one cable connect to the receive leads of the other cable, and vice versa. There are two ways to be sure the cables connect properly. The first is to use a crossover cable, which connects the transmit and receive leads of one cable to the appropriate leads on the other cable. The other way (available on RTNI-2T1 and RTNI-2E1 boards only) is to set the mode of the trunk using hardware jumpers, then connect the trunks using a regular cable. Set one trunk to user mode and the other trunk to network mode, as described in your hardware installation guide. For normal operation, both trunks should be set to user mode. Figure 21 shows trunks A and B connected by a cable.

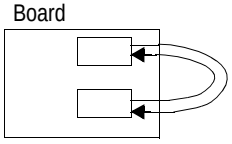

Figure 21. Connecting Trunks A and B in E1

Troubleshooting

If a line function returns an error code, use BrktGetLastError () to determine what caused the error. The two most common errors are lost synchronization or a protocol error.

- In a synchronization error, BrktGetLastError () returns BRKT_ERROR_CODE (UNEXP_NET_ERROR).
- In a protocol error, BrktGetLastError() returns BRKT_ERROR_CODE(IO_DEVICE).

Handling Synchronization Errors

When a trunk loses synchronization, any bits the trunk receives are invalid. After a loss of synchronization, terminate all calls and check the trunk status using QUERY_CARRIER_STAT. When the trunk is synchronized with the network, the application can proceed.

QUERY_CARRIER_STAT returns information about the current alarm and synchronization status of the line in the structure *RTNI_E1carrierStatus_s*. It also returns any errors that have occurred since the last time the application called that function. This information can help you measure the quality of the link to the network. If the data returned indicates the line is not synchronized or that alarms are present and the condition persists for longer than a few hundred milliseconds, the application should terminate all calls and issue an error message.

RTNI_E1carrierStatus_s returns the following information about the line:

Alarm — Indicates whether there are any alarms on the line. There are three alarm types:

Frame Zero — Timeslot 16 of frame zero contains an alarm bit. The remote end sets this bit to signal an alarm.

Timeslot 16 All Ones — Indicates that all bits in timeslot 16 are ones. Since timeslot 16 carries signaling bits for all channels, a timeslot 16 all ones means that the signaling bits are not valid.

This alarm is usually accompanied by a loss of synchronization, since the framing information on timeslot 16 is necessary to identify the E1 multiframe.

All Ones — Indicates that all bits received are ones. In this situation, all framing information is lost. An all ones signal is usually sent as a test pattern to verify the quality of the data link.

Slips — Buffer slips indicate that the E1 board is not reading and writing data to the line at the same rate as the remote end. Usually, an incorrect clock setting or faulty clock source are responsible for buffer slips. For more information about setting the clock, see *Setting the Clock* on page 100.

Sync

The synchronization field indicates whether or not the network and E1 board are synchronized. If they are not synchronized, it means that the board is not receiving framing information. This can happen if there is no signal on the line, if the board is receiving an invalid or test signal, or if the framing mode is not set properly. Be sure that you have set the appropriate clock mode, CRC mode, and framing method, as described in *Configuring the E1 Environment* on page 100.

If changing the clock mode, CRC mode, or framing method doesn't restore synchronization, the cable might be improperly installed. Use a remote and local loopback to isolate whether the problem originates at the CO or your system. For more information about loopbacks, see *Using Loopbacks* on page 140.

CRCerrorCount

A CRC error indicates that the CRC calculated for one particular submultiframe did not match the value transmitted by the far end. Since the far end and your board perform the same calculation, the data was corrupted during transmission.

One reason for CRC errors is if the signal level received at your premises is low, so some one bits were interpreted as zero bits. Be sure the equalization parameters on the board are set incorrectly for the cable length, as described in the hardware installation card for your E1 board. Improper equalization settings can cause low signal levels. Also, check the signal level provided by the carrier and increase it if possible.

CRC error indications are only meaningful if your system is synchronized with the far end. Buffer slips have no effect on CRC calculations.

Simux

The Simux bits report the result of the CRC comparison at the far end. If there are too many errors, the E1 board might be transmitting poor quality of data or there might be a problem with the cables.

The Simux values are only meaningful if you are receiving a clear signal, without synchronism errors or excessive CRC errors. If you receive a clear signal, it means that the lines from the CO to the E1 board are good, so any cabling problems are from the E1 board to the CO.

Handling Protocol Errors

A protocol error occurs when the signaling bits read from the line don't match what was expected at that time. For example, a function detects call answer while monitoring for disconnect. An unexpected event is not necessarily an error, since there are times when more than one event could happen. However, the software returns an error if any event other than the one the function is waiting for takes place.

The best way to recover from a protocol error is to call RHT_DISCONNECT to set the local end to idle, then call RHT_WAIT_IDLE and wait for it to return. When that happens, both ends are in an idle state and are ready to send or receive calls. When the line is idle, the application can continue. Call RHT_GET_STATUS for more information about what caused the protocol error. RHT_GET_STATUS returns information about the protocol and compiler version as well as the termination code and line function information in the structure *RTNI_lineStatus_s*.

The 'TermType', 'RawPattern', and 'Function' fields in *RTNI_lineStatus_s* contain information to help you handle protocol errors. Providing these fields to technical support greatly reduces the time it takes to diagnose a problem.

TermType

'TermType' contains the terminating code for the last E1 line function. If the function is still running, 'TermType' contains an intermediate message or the protocol code for the previous function. Some protocol codes indicate that the previous function terminated correctly. For example, a protocol code might indicate that the RHT_WAIT_LINE_ON function terminated because it received a call.

Other protocol codes indicate a protocol error. The *RealCT Direct API Reference Manual* contains a complete list of protocol codes and information about how to interpret these values for different functions.

The protocol codes BAD_STATE_ERROR and SIGNALING_BIT_ERROR are common in protocol errors.

BAD_STATE_ERROR

A BAD_STATE_ERROR indicates that the function could not execute because the line was in the wrong state for that function. For example, 'TermType' would report a bad state error if the application tries to wait for a call when the line is not idle. In a bad state error, the CO's interpretation of the previous signal on the line probably didn't match your application's.

A bad state error could also occur if the application expects a certain event that does not occur, such as a seizure acknowledgment after seizing the line. Sometimes the signal is not sent because the CO and the application are in different states. This could happen if the application sends a signal that does not meet a minimum time requirement, so the CO does not recognize the signal. The application would be in one state, having sent the signal, but since the CO did not recognize the signal, it would not change states. Guard times generally prevent this situation.

The field 'RawPattern' contains the signaling bits being sent and received on the line. This information can help determine why a bad state error occurred, but doesn't explain why the situation arose.

SIGNALING_BIT_ERROR

A signaling bit error occurs when a bit pattern that does not belong in the protocol was present on the line. The invalid bit pattern must be present for longer than the debouncing and deglitching timings in order to generate this error.

If you receive a signaling bit error, compare the bit pattern in *RTNI_lineStatus.RawPattern* to the ones expected by the protocol in case your carrier is using a variation on the protocol. Call the carrier for more information about their protocol, or use AccuSpan to get more information about the signaling.

If you frequently get a signaling bit error that isn't related to any particular event, you might have a problem with the connection. In this case the application should monitor the status of the carrier. If the number of CRC errors or buffers slips is high, the line quality is poor. Check the gain your carrier is set to provide, the length of cable, and the equalization parameters set on the hardware.

RawPattern

'RawPattern' contains the received and transmitted signaling bits. The upper byte contains the received pattern while the lower byte contains the transmitted pattern. Each of these bytes contains only four significant bits. These bits contain the value for signaling bits A, B, C, and D. Bit A is the most significant bit, while bit D is the least significant bit.

For example, if 'RawPattern' contains 0x0F05, then the receive pattern is 0x0F and the transmit pattern is 0x05.

Receive =	0x0F=1111, and A=B=C=D=1
Transmit =	0x05=0101, and A=C=0 and B=D=1

If you are transmitting signaling bits when using AccuSpan in signaling mode, follow the same conventions.

Function

'Function' shows which function is currently running or which function terminated last. You need to know the function name in order to understand the terminating code listed in 'TermType'.

The 'Function' field is particularly useful in finding out the cause of BRKT_ERROR_CODE(BUSY). This error means that the line function specified by the 'Function' field was already running when the application called another line function. BRKT_ERROR_CODE(BUSY) is usually a result of a programming error.

Using Loopbacks

Loopback configurations loop data back to where it originated. These configurations help troubleshoot where errors occur. There are several forms of loopbacks, depending on what you want to test.

Set software loopbacks using CONFIG_CARRIER as described in *Configuring the Carrier* on page 105. There are two software loopbacks:

- Test whether the board's receiver and transmitter work properly using a local loopback. In this mode, the board receives the same data it transmits without passing through any cabling. To do a local loopback, set the loopback mode in CONFIG_CARRIER to local.
- Test the cabling and verify that the CO can send and receive valid data using a remote loopback. In this mode, the board immediately transmits all data it receives back to the CO. To do a remote loopback, set the loopback mode in CONFIG_CARRIER to remote.

Figure 22 shows remote and local loopbacks.

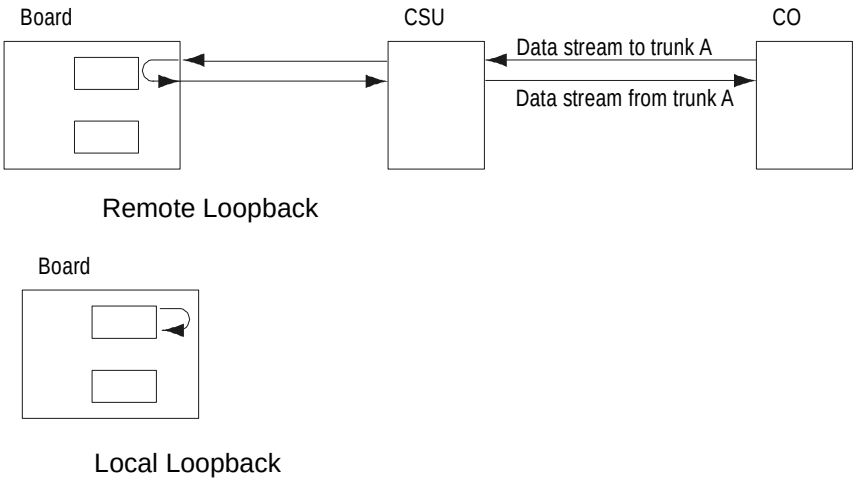

Figure 22. Remote and Local Loopbacks in E1

Test individual regions of the cabling using a cable to connect transmit and receive leads at points along a trunk. This mode loops data back to either the board or CO depending on which end transmits the data and how you set up the loopback. Figure 23 shows a loopback using a cable to physically connect the transmit and receive leads along trunk A.

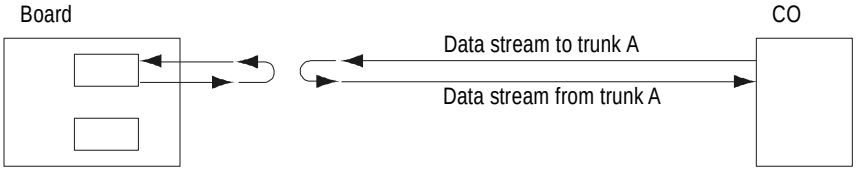

Figure 23. Cabling Loopbacks in E1

5
MVIP-90

This chapter describes how to develop applications for systems using the Multi-Vendor Integration Protocol (MVIP). It discusses the MVIP-90 software and hardware standard used by the MVIP-compliant RDSP, Vantage VPS, Vantage VRS, and RTNI boards.

This chapter includes the following topics:

- Defining MVIP-90
- Working with MVIP-90 Data Streams
- Understanding Framing
- Configuring Boards in the MVIP Bus
- Configuring the MVIP-90 Clock
- Mapping MVIP-90 Resources
- Enabling or Disabling Resources
- Switching Calls through the MVIP-90 Bus

Defining MVIP-90

Multi-Vendor Integration Protocol (MVIP) provides communications standards that allow boards in a PC to communicate with each other. MVIP works independently of other computer buses. It can work in a PC, between PCs, in a PBX, and in other computer-based hardware.

The MVIP bus provides a way to interconnect telephony resources, even if those resources are provided by different vendors. With voice, fax, video, and automatic speech recognition cards connected in a single bus, you can develop fully-integrated computer telephony applications.

MVIP is comprised of two main parts: the MVIP bus, and the MVIP switch block. The MVIP bus provides the physical connection between boards. The MVIP switch block provides the mechanism for data switching between telephony resources.

Only the RTNI series network boards and Vantage PCI boards provide a switch block and can make physical connections between resources. RTNI series boards do not provide VP resources, but they can switch calls to VP resources on other boards. The Vantage VPS, Vantage VRS, and RDSP/xx000 boards do not have a switch block. They provide VP resources over the MVIP bus.

Brooktrout supports two levels of MVIP: MVIP-90 and MVIP-95. RDSP, Vantage VPS, Vantage VRS, and RTNI boards all use the MVIP-90 software and hardware standard. Applications written using the MVIP-90 software standard only support MVIP-90 hardware. The Vantage PCI series boards support the MVIP-95 software standard. The MVIP-95 standard supports both MVIP-90 and H.100 hardware.

The API for Windows operating systems has functions for both the MVIP-90 and MVIP-95 standards. Be sure you use the functions appropriate for the boards in your system.

- Use MVIP-90 functions for RTNI, RDSP, Vantage VPS or Vantage VRS boards.
- Use MVIP-95 functions for Vantage PCI and RealBLOCs PCI boards.

For more information about the MVIP-95 standard, see Chapter 6, *MVIP-95,* on page 183.

Working with MVIP-90 Data Streams

Understanding MVIP-90 Architecture

The MVIP-90 bus supports eight bidirectional 2.048 Mb/s streams for a total of 16 data streams, as shown in Figure 24. These streams carry data to and from MVIP resources. Each MVIP stream has 32 64-Kb/s time division multiplexed slots, each of which supports a single resource (32 slots x 64 Kb/s/slot = 2.048 Mb/s). Each slot within a specific stream is called a timeslot.

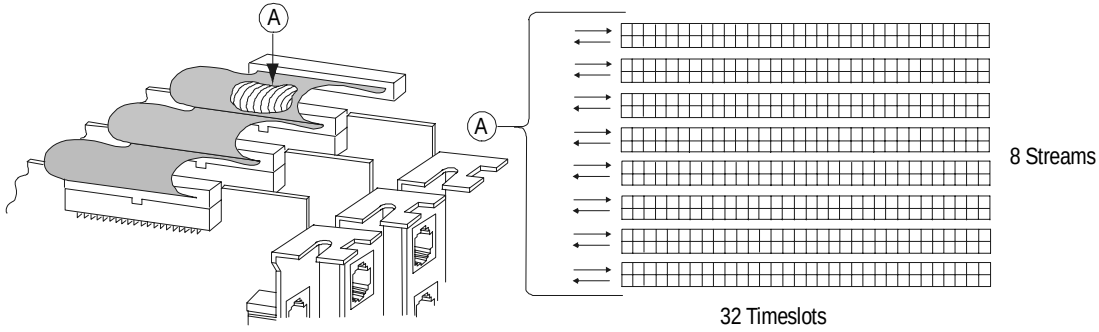

Figure 24. Data Streams in an MVIP-90 Bus

Each 2.048 Mb/s stream supports all channels on a voice processing resource card, one resource per timeslot. You set which stream a board uses through either hardware or software, depending on the board. The single stream serves as a data highway between the network interface and resource board assigned to that stream. Timeslots in the stream are like mailboxes, carrying information to the resource mapped to that address. See *Mapping MVIP-90 Resources* on page 155 for more information about specifying MVIP streams and timeslots.

Boards with switching capability such as the RTNI or Vantage PCI series boards are not assigned to a particular stream. These boards can place or retrieve data on any stream or timeslot.

An application specifies the streams and timeslots to switch data between resources. The RTNI or Vantage PCI switch block then performs the switching.

For example, in a voice mail system, the application instructs the MVIP switch block to route data from the stream and timeslot associated to the inbound line where the call was received to the stream and timeslot used by the voice processing resource that plays the pre-recorded welcome message. Then, to form a bidirectional connection, the application instructs the switch block to make the reverse connection: from the VP resource back to the line resource.

Numbering MVIP Streams

Of the 16 data streams, the first eight are input and the second eight are output. Together, the first input (0) and first output (8) streams form the bidirectional stream 0, as shown in Figure 25. The second input (1) and second output (9) form the bidirectional stream 1, and so on through the eighth input (7) and eighth output (15), which form the bidirectional stream 7. Although the driver uses 16 streams, the application only sees the eight bidirectional streams.

Input and output streams are designated as DSix and DSox where DS stands for data stream; i stands for in; o stands for out; and x stands for the number of the bidirectional stream, 0 through 7. For example, an application would refer to the input and output streams that make up the seventh bidirectional stream as DSi6 and DSo6. The driver interprets these stream assignments as streams 6 and 14, as shown in Figure 25.

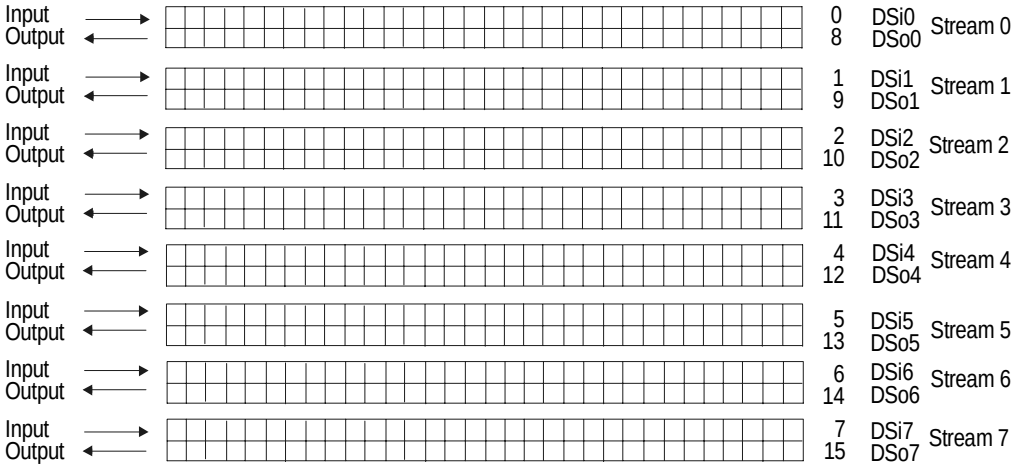

Figure 25. Data Stream Numbering

The terms ‘input’ and ‘output’ are used in reference to the resource board. A DSix stream carries data from the network to the resource board (input to the resource) while DSox streams carry data from the resource to the network (output from the resource). Network boards transmit data on input streams and receive data on output streams.

Understanding Framing

Data in DSix or DSox streams are formatted into frames. Each frame contains 8 bits of information for each of the 32 timeslots, as shown in Figure 26. Each 8 bit timeslot has a period of 125 µs, for a total of 8000 timeslots per second (8 bits/timeslot x 8000 timeslots/second = 64 Kb/s).

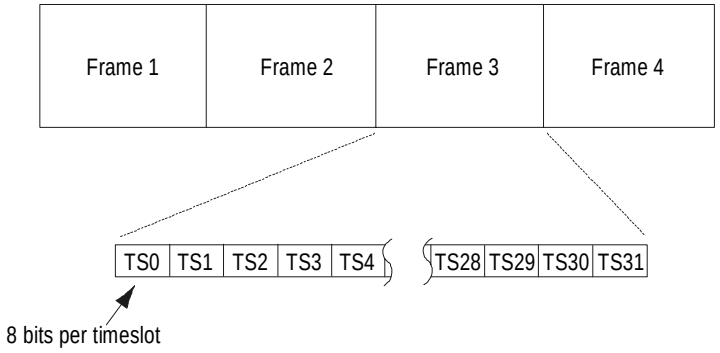

Figure 26. Timeslots in an MVIP Frame

The MVIP-90 clock carries timing information to synchronize the frames. The clocking information travels on a dedicated pair of wires rather than taking up space in the stream. Since streams are devoted to transmitting data rather than signaling information, they are called clear channels. For more information on setting the MVIP clocks, see *Configuring the MVIP-90 Clock* on page 150.

Configuring Boards in the MVIP Bus

There are several steps to configuring your MVIP bus. The following sections in this chapter discuss these steps in more detail.

1. Configure the MVIP clock.

   The MVIP bus uses a clock to synchronize frames. Set the clock for all boards in the bus except Vantage VRS or RDSP/xx000 boards. These boards have no clock and cannot drive the MVIP clock.

2. Map resources to streams and timeslots.

   Resource boards are mapped to a specific stream on the MVIP bus. You set the stream assignment either during hardware or software configuration. Resources are mapped to specific timeslots on the assigned stream. Although network boards such as the RTNI or Vantage PCI boards are not mapped to a specific stream, their internal resources are assigned to timeslots on internal streams.

   Resource mapping for the Vantage PCI and RTNI boards is pre-configured. This step is only needed if you have Vantage VPS, Vantage VRS, or RDSP/xx000 boards.

3. Enable or disable resources.

   By default, resources on Vantage VPS or Vantage PCI boards are not available to the MVIP bus. Enable these resources before performing switching functions.

4. Establish connections.

   When you switch calls, you establish connections between a board's resources and the MVIP bus.

Configuring the MVIP-90 Clock

Understanding Clocking Signals

Boards connected in an MVIP-90 bus use an 8-kHz clock signal to synchronize frames.

The MVIP-90 bus uses the following clocks:

- /F0 is the primary 8-kHz framing signal
- /C4 is the bus 4.096 MHz clock
- /C2 is the bus 2.048 MHz clock
- SEC8K is a secondary 8-kHz signal

An RTNI board in the system drives the /F0 clock, which other boards use as a reference. You can also specify that boards use SEC8K as a backup signal. SEC8K acts as a fallback in case the /F0 signal fails. You do not configure either the /C4 or /C2 clocks.

Setting the Clock for T1 or E1 Boards

T1 and E1 boards receive T1 and E1 data in frames. These frames must be synchronized with the network clock. For this reason, if a system contains an T1 or E1 board, that board must set the MVIP clock from the signal received on the first or second network trunk. All other boards in the system extract their clock from the MVIP bus.

Setting the Clock for RTNI-ATSI Boards

An RTNI-ATSI board sets the MVIP clock only if an T1 or E1 is not present in the system. The RTNI-ATSI uses its internal oscillator to generate the clock signal.

Setting the Clock for Vantage PCI Boards

Vantage PCI boards set the MVIP clock only if an T1 or E1 is not present in the system. The Vantage PCI board uses its internal oscillator to drive the clock. Set the Vantage PCI board's clock using MVIP-95 functions described in Chapter 6, *MVIP-95,* on page 183.

Setting the Clock for Vantage VPS Boards

Vantage VPS boards set the MVIP clock only if an T1 or E1 is not present in the system. The Vantage VPS board takes the clock from the internal oscillator. Set the Vantage VPS board's clock using the RHT_SET_CLOCK function.

Setting the Clock for RDSP and Vantage VRS Boards

RDSP and Vantage VRS boards cannot set the MVIP clock. They must retrieve the clock signal from the MVIP bus.

Setting the Clock Parameters

Set the MVIP clock during driver installation and configuration using the Configuration Wizard. For specific information about how to set the MVIP clock during driver configuration, see the *RealCT API Installation and Configuration Guide*.

If you need to change the clock parameters after software installation, run the Configuration Wizard or use the functions CONFIG_CLOCK for RTNI boards, or RHT_SET_CLOCK for Vantage VPS boards, as described in the *RealCT Direct API Reference Manual*.

When you use CONFIG_CLOCK, first configure the board setting the MVIP clock. Then configure any other RTNI, Vantage PCI, or Vantage VPS boards in the system.

For each board, specify one of the following as the clock source:

CLOCK_REF_NET1	Drives the clock from the first network trunk.
CLOCK_REF_NET2	Drives the clock from the second network trunk.
CLOCK_REF_OSC	Drives the clock from the internal oscillator.
CLOCK_REF_MVIP	Takes the clock from the /F0 signal.
CLOCK_REF_SEC8K	Takes the clock from the SEC8K signal. We don't recommend taking the primary clock from the SEC8K signal.

For each board, you also set whether it drives the SEC8K as a backup signal as follows:

SEC8K_NOT_DRIVEN	Board does not drive the SEC8K clock.
SEC8K_DRIVEN_BY_OSC	Board drives the SEC8K clock from the internal oscillator.
SEC8K_DRIVEN_BY_NET1	Board drives the SEC8K clock from the first network trunk.
SEC8K_DRIVEN_BY_NET2	Board drives the SEC8K signal from the second network trunk.

Setting up the System

The following examples show how to set the MVIP clock for common scenarios:

- If you have a system with a T1, an RTNI-ATSI, one Vantage VPS, and one Vantage VRS, you would set the clock in the following order:
  a. Set the T1 board to set the MVIP clock based on the first network trunk and to retrieve the SEC8K signal as a backup:

     `CLOCK_REF_NET1 | SEC8K_NOT_DRIVEN`

     (use the RHT_SET_CLOCK function)
  b. Set the RTNI-ATSI board to retrieve the clock from the MVIP bus and to drive the SEC8K clock based on its internal oscillator:

     `CLOCK_REF_MVIP | SEC8K_DRIVEN_BY_OSC`

     (use the RHT_SET_CLOCK function)
  c. Set the Vantage VPS board to take the clock from the MVIP bus:

     `CLOCK_REF_MVIP`

     (use the RHT_SET_CLOCK function)
  d. There is no need to set the Vantage VRS board clock
- If you have a single T1 or E1 board used in loopback mode (trunk A connected to trunk B)

  `CLOCK_REF_OSC | SEC8K_NOT_DRIVEN`

  (use the RHT_SET_CLOCK function)

- If you have multiple T1 or E1 boards connected in loopback mode:
  a. Set the first T1 or E1 board to drive the MVIP clock from the internal oscillator and drive SEC8K from the first network trunk:

     `CLOCK_REF_OSC | SEC8K_DRIVEN_BY_NET1`

     (use the RHT_SET_CLOCK function)
  b. Set all other boards to take the clock from the MVIP bus:

     `CLOCK_REF_MVIP | SEC8K_NOT_DRIVEN`

     (use the RHT_SET_CLOCK function)

Example 7 shows how to set the clock for an T1 board using CONFIG_CLOCK. In Example 7, the T1 board sets the primary and SEC8K clock from the first network trunk. This code could also be used for an E1 board by changing the name of the device.

Example 7. CONFIG_CLOCK Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE BoardHandle;        /* T1 board device handle */
   BOOLEAN IoctlResult;            /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;      /* Bytes returned from IOCTL call*/
   USHORT ClockValue;

     /* Open T1 board device */
   BoardHandle = BrktOpenDevice (BRKT_DEVICE_T1_BOARD, 0);

     /* Configure MVIP clock. Synchronous call. */
   ClockValue = CLOCK_REF_NET1 | SEC8K_DRIVEN_BY_NET1;

   IoctlResult = BrktDeviceIoControl (
         BoardHandle,
         CONFIG_CLOCK,
         &ClockValue,              /* Buffer to driver */
         sizeof(ClockValue),
         NULL,
         0,
         &BytesReturned,
         NULL);                    /* Wait until I/O is complete */

   if (!IoctlResult)
         printf ("CONFIG_CLOCK failed: BrktGetLastError = %d\n",
               BrktGetLastError (BoardHandle));

   BrktCloseDevice(BoardHandle);
}
```

Mapping MVIP-90 Resources

RDSP, Vantage VPS, and Vantage VRS boards each use a single data stream in the MVIP bus to send and receive data. Within that stream, the driver assigns VP and line resources on these boards to use a specific timeslot.

RTNI and Vantage PCI boards use internal streams for their internal line and VP resources. Resources are pre-configured to specific timeslots on those streams.

Mapping RTNI Resources

Mapping RTNI Streams

The RTNI series boards map the data received or transmitted on its lines to data streams called internal streams. In T1 and E1 boards, Trunk 0 uses internal stream 16 to transmit and receive and Trunk 1 uses internal stream 18 to transmit and receive. RTNI-ATSI boards only use stream 16.

Timeslots on these internal streams can be connected to timeslots on any stream in the MVIP bus. When you send a command to an MVIP switch block, you specify a board handle. The switch block then switches data from the internal streams on the board referred to by the board handle to the MVIP bus.

Mapping RTNI Internal Resources

Line resources on RTNI boards are assigned to timeslots on internal streams sequentially, starting with 0. For example, on trunk 0, the resource 0 is assigned to timeslot 0 on stream 16; resource 1 is assigned to timeslot 1 on stream 16, and so on through resource 23, which is assigned to timeslot 23 on stream 16.

On E1 boards, only the 30 E1 channels that carry data are mapped to the internal streams (E1 timeslots 1-15, 17-31). These are mapped to internal timeslots 0-29. E1 timeslots 0 and 16, which carry signaling data, are not mapped to internal timeslots.

Mapping RDSP/xx000, Vantage VRS, and Vantage VPS Resources

Mapping Boards to Streams

RDSP/xx000, Vantage VRS, and Vantage VPS boards do not use internal streams. Resources on these boards are mapped directly to streams and timeslots on the MVIP bus.

RDSP/xx000

Set the MVIP stream assignment for RDSP/xx000 series boards using hardware jumpers. These jumpers are factory configured to stream 6. To change the stream assignment, follow the instructions in the hardware installation guide that came with your board. Be sure each RDSP/xx000 is assigned to a unique stream on the MVIP bus.

Vantage VRS

Set the MVIP stream assignment for Vantage VRS boards using the Configuration Wizard during driver configuration. If you need to change the settings after driver configuration, run the Configuration Wizard again or use the RHT_CONFIG_MVIP function. All boards use stream 6 by default. Be sure each Vantage VRS is assigned to a unique stream on the MVIP bus.

Each DSP on a board uses the same stream. If you attempt to assign DSPs on one board to different streams, all DSPs on the board use the last stream you select.

Vantage VPS

Set the MVIP stream assignment for Vantage VPS boards using the RHT_CONFIG_MVIP function. When you call RHT_CONFIG_MVIP, set the stream assignment for the line and the VP resources separately. Set both the VP and line resources to use the same bidirectional stream. For example, if you set VP resources to use stream 6 (an input stream), set the line resources to use stream 14 (the corresponding output stream). Be sure each Vantage VPS is assigned to a unique stream on the MVIP bus.

Mapping Resources to Timeslots

RDSP, Vantage VPS, and Vantage VRS boards use one timeslot on their assigned MVIP stream per resource. Table 26 shows how resources are assigned to timeslots. VP indicates the VP resource number and TS indicates the timeslot used by that resource.

Table 26. Resource Mapping

	VP resource 1	VP resource 2	VP resource 3	VP resource 4
DSP1	VP: 0 TS: 1	VP: 1 TS: 9	VP: 2 TS: 17	VP: 3 TS: 25
DSP2	VP: 4 TS: 2	VP: 5 TS: 10	VP: 6 TS: 18	VP: 7 TS: 26
DSP3	VP: 8 TS: 3	VP: 9 TS: 11	VP: 10 TS: 19	VP: 11 TS: 27
DSP4	VP: 12 TS: 4	VP: 13 TS: 12	VP: 14 TS: 20	VP: 15 TS: 28
DSP5	VP: 16 TS: 5	VP: 17 TS: 13	VP: 18 TS: 21	VP: 19 TS: 29
DSP6	VP: 20 TS: 6	VP: 21 TS: 14	VP: 22 TS: 22	VP: 23 TS: 30
DSP7	VP: 24 TS: 7	VP: 25 TS: 15	VP: 26 TS: 23	VP: 27 TS: 31
DSP8	VP: 28 TS: 0	VP: 29 TS: 8	VP: 30 TS: 16	VP: 31 TS: 24

Table 27 shows the VP resource number in the top row and their associated timeslots in the bottom row.

Table 27. VP Resources and Their Associated Timeslots

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	9	17	25	2	10	18	26	3	11	19	27	4	12	20	28	5	13	21	29	6	14	22	30	7	15	23	31	0	8	16	24

Since Vantage VPS boards only have one DSP, they only use timeslots 1, 9, 17, 25 by default. RDSP/xx000 and Vantage VRS boards use a varying number of timeslots, depending on the number of resources that board provides.

Although RDSP/xx000, Vantage VPS, and Vantage VRS resources use the timeslots shown in Table 27, you can offset the timeslot assignments for Vantage VRS and Vantage VPS boards when you call RHT_CONFIG_MVIP to map boards to a stream.

The default timeslot assignment shown in Table 26 and Table 27 uses an offset of '1'. Notice that resource 0 uses timeslot 1. Using an offset of '4', resource 0 would use timeslot 4. When designing an application, do not confuse resource numbers with timeslot numbers. For example, if you switch a call from an RTNI board to timeslot 1 of an RDSP/24000, you would use VP resource 0 not VP resource 1 to record that call. This is because timeslot 1 corresponds to resource 0.

The equation for timeslot assignment is as follows:

timeslot = [(VP/4) + offset]%8 + [(VP%4) *8]

where the default offset is 1.

In applications, the easiest way to determine the timeslot assignment is with an array:

```
int mapping []=          1, 9,17,25
                         2,10,18,26
                         3,11,19,27
                         4,12,20,28
                         5,13,21,29
                         6,14,22,30
                         7,15,23,31
                         0, 8,16,24

timeslot=mapping[VP%32]
```

For VPS boards you call RHT_CONFIG_MVIP twice: once to set stream and timeslot assignments for VP resources and once to set stream and timeslot assignments for line resources. If you set both line and VP resources to use the same timeslot offset and stream, they are connected in channels through the MVIP bus.

For example, if you set line and VP resources to use an offset of 1, the first line resource will use the timeslot 1 of the DSox stream and the first VP resource will use timeslot 1 of the DSix stream. If you set the line and VP resources to have different offsets, the network board controls how the line and VP resources are used.

Enabling or Disabling Resources

Boards that provide both line and VP resources, such as the Vantage VPS, have their resources connected to form channels. In this state, a call that comes in on line 0 is automatically processed by VP 0, as shown in Figure 27. The resources are not available to the MVIP bus.

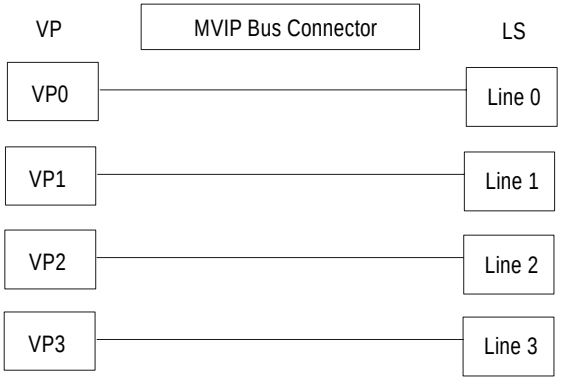

Figure 27. Disabled VP and LS Resources

To make these resources available to the MVIP bus, enable the resources using RHT_CONFIG_MVIP and RHT_MVIP_SETTING. Figure 28 shows the resources enabled.

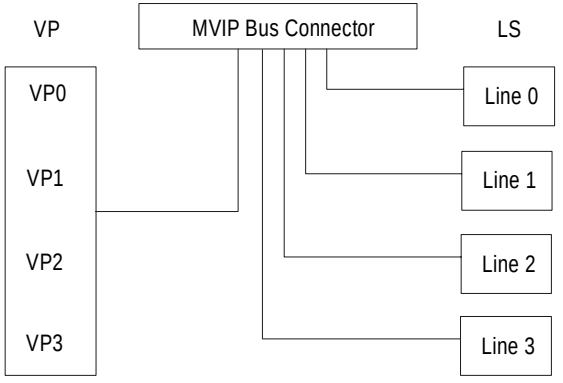

Figure 28. Enabled VP and LS Resources

Enable resources before you set the clock or switch data to resources on the Vantage VPS board. There are two ways to enable or disable line and VP resources.

- When you call RHT_CONFIG_MVIP to set stream and timeslot assignments, enable or disable resources.
- If you have already set the stream and timeslot assignments, enable and disable resources using RHT_MVIP_SETTING.

The two functions enable and disable resources in the same way. When you enable VP resources, you enable all resources at once. You can enable line resources individually. To make all resources available to the MVIP bus, call either RHT_CONFIG_MVIP or RHT_MVIP_SETTING twice: once for the line resources and once for the VP resources.

You can also enable just the VP or just the line resources. You would enable line resources individually if you want to bridge a call between the Vantage VPS line and a line on a network board. You would enable VP resources if a network board needs to switch data to a VP resource. All VP resources are enabled even if you just need to use a single resource.

Example 8 shows how to use RHT_CONFIG_MVIP to configure LS lines on a Vantage VPS board. In this example, the line resources are mapped to use stream 8 (the first transmit stream). The first resource is mapped to timeslot 0.

Example 8. RHT_CONFIG_MVIP Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE BoardHandle;        /* DSP board device handle */
   BOOLEAN IoctlResult;            /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;      /* Bytes returned from IOCTL call*/
   struct MVIPconfig_s Config;

     /* Open DSP device */
   BoardHandle = BrktOpenDevice (BRKT_DEVICE_RDSP_BOARD, 0);

     /* Configure Vantage LS lines to MVIP. Synchronous call. */
   memset (&Config,0,sizeof (Config));
   Config.Resource = RDSP_LINE;
   Config.Stream = 8;              /* LS lines are connected to
                                      streams 8-15 in the opposite
                                      direction of VP devices */
   Config.Timeslot = 0;

   IoctlResult = BrktDeviceIoControl (
         BoardHandle,
         RHT_CONFIG_MVIP,
         &Config,                  /* Buffer to driver */
         sizeof(Config),
         NULL,
         0,
         &BytesReturned,
         NULL);                    /* Wait until I/O is complete */

   if (!IoctlResult)
         printf ("RHT_CONFIG_MVIP failed: BrktGetLastError = %d\n",
                 BrktGetLastError (BoardHandle));

   BrktCloseDevice (BoardHandle);
}
```

Switching Calls through the MVIP-90 Bus

When a computer telephony system processes a call, it connects the inbound or outbound line with the appropriate resource, such as VP or another line. That other resource can be on the same board, or on a different board in the system. The process of routing data between resources on different boards is called switching. All line resources have a CODEC, which codes/decodes the analog voice signal to digital data. It is the digital representation of the analog signal that a switch block switches between resources.

Only boards with a switch block, such as the RTNI or Vantage PCI boards, can perform switching functions. Although the RDSP/xx000, Vantage VPS, and Vantage VRS boards can connect to the MVIP bus, they do not have a switch block and can not switch data.

Boards with switching capabilities have access to all MVIP streams and timeslots. The Vantage VPS, Vantage VRS, and RDSP/xx000 boards, on the other hand, can only access their assigned streams. Data placed on the DSix stream goes directly to the resources of the board assigned to that stream. The resource places data on the DSox stream.

Boards with switching capability use internal data streams for their line or VP resources. These internal streams carry data from lines coming in to the RTNI board. They do not connect to the MVIP bus and cannot be accessed by resources outside the board they are internal to. When you make a connection between an internal resource and the MVIP bus, you specify a board handle so the application knows which switch block transfers data from its internal stream.

Throughout this section, boards connected in an MVIP bus are depicted as shown in Figure 29.

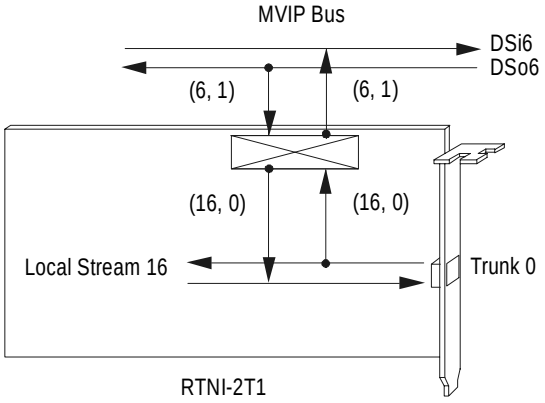

Figure 29. A Board in an MIVP Bus

In Figure 29, the arrows above the boards represent streams in the MVIP bus. Streams running within the board to and from internal line or VP resources represent local streams. The rectangle with an X is the switch block. Arrows connecting MVIP bus and local resources through the switch block show which streams the switch block is switching data between. Numbers beside the switching arrows define the stream and timeslot that the switch block is switching data between.

Establishing Connections

Establish or disconnect an MVIP connection using the function SET_OUTPUT. When the application calls SET_OUTPUT, it specifies the stream and slot to connect as output and the stream and slot to connect as input. It also specifies the timeslot mode.

You can set the *OutputParam_s.Mode* field to either disable mode, connect mode, or pattern mode.

- When *OutputParam_s.Mode* is in CONNECT_MODE, use SET_OUTPUT to connect the timeslot defined by *OutputParam_s.OutputStream* and *OutputParam_s.OutputSlot* to the timeslot defined by *OutputParam_s.InputStream* and *OutputParam_s.InputSlot.* This connection allows the timeslot defined by *OutputStream, OutputSlot* to receive data from the *InputStream, InputSlot* timeslot. To make a bidirectional connection, swap the input and output timeslots in the *OutPutParam_s* data structure and call SET_OUTPUT again.
- When *OutputParam_s.Mode* is in PATTERN_MODE, SET_OUTPUT repeatedly transmits *OutputParam_s.Message* to the timeslot defined by the *OutputParam_s.OutputStream* and *OutputParam_s.OutputSlot*.
- When *OutputParam_s.Mode* is in DISABLE_MODE, use the function SET_OUTPUT to disconnect the timeslot defined by *OutputParam_s.OutputStream* and *OutputParam_s.OutputSlot* from the MVIP bus and internal streams. You cannot disconnect an ATSI connection using DISABLE_MODE. Instead, use PATTERN_MODE and transmit a silence.

When you use SET_OUTPUT to make a connection, you automatically break any previous connection that used the same resource as output. This is because a resource can only have one input source. For example, if an T1 board is switching data to the second VP resource on a Vantage VRS, and a second RTNI board also switches data to the second VP resource, the first connection will be broken.

If two resources are both using the same MVIP bus timeslot as output, the connection is not broken. In this case, both resources input data to the same timeslot, but the two sources cancel each other, resulting in no audio output.

Using Stream Numbers

When calling any MVIP function, you refer to streams by a 0-15 numbering scheme rather than by their DSix and DSox numbers, as shown in Table 28. When you use streams in their conventional direction, the reference number is the same as the stream number. When you use streams in the reverse direction, you use the stream number plus eight. For examples of when you would use a stream in a reverse direction, see *Connecting Line Resources* on page 175. The driver use the number assignment and whether the stream is being used as input or output to select the appropriate DSix or DSox stream.

Table 28. MVIP Stream Numbering

	Used as	
Stream	**Input Stream**	**Output Stream**
DSi0	8	0
DSi1	9	1
DSi2	10	2
DSi3	11	3
DSi4	12	4
DSi5	13	5
DSi6	14	6
DSi7	15	7
DSo0	0	8
DSo1	1	9
DSo2	2	10
DSo3	3	11
DSo4	4	12
DSo5	5	13
DSo6	6	14
DSo7	7	15

Keep in mind that the conventional direction uses the resource board as a reference, so resource boards transmit on the DSo streams and receive on the DSi streams. When you call SET_OUTPUT, you define input and output from the network board's perspective, since the switch block is part of the network board. From the network board's perspective, the conventional direction uses DSi as output (input to the resource board) and DSo as input (output from the resource board).

For example, if you transfer a call from a T1 to a Vantage VRS using stream 1, you would use DSi1 as *OutputStream* (input to the VRS) in SET_OUTPUT, and DSo1 as *InputStream* (output from the VRS). As Table 28 shows, using DSi1 as output and DSo1 as input, you would use the stream assignment 1 for both *OutputStream* and *InputStream*.

Connecting Local Resources

An RTNI board can connect a line to a remote resource through the MVIP bus, or connect two lines on the same board. For example, the T1 board in Figure 30 is connecting an incoming call from its second line (16, 1) with its fourth line (16, 3). In this example, the T1 board's switch block does not contact the MVIP bus. When you call SET_OUTPUT to make this connection, you would specify the board handle of the T1 board you want to use.

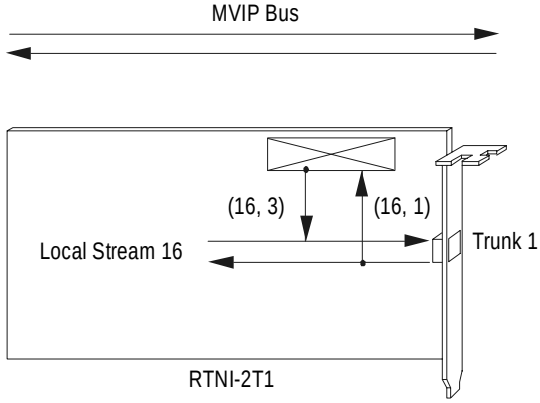

Figure 30. A Call Transfer Between Two Internal Timeslots

Make the half-duplex connection shown in Figure 30 as follows:

```
SET_OUTPUT                    (INPUT {16,1}
                              OUTPUT {16,3})
```

Connecting a Call to a Resource Board

When an RTNI board receives a call, it can transfer that call to a VP resource on a resource board such as a Vantage VRS or RDSP/24000. To make the connection, the RTNI board's switch block switches the call from its internal line to the MVIP stream assigned to the resource board. Figure 31 shows an T1 board connecting a call from its first line resource to the first resource on a Vantage VRS that's assigned to use stream 6.

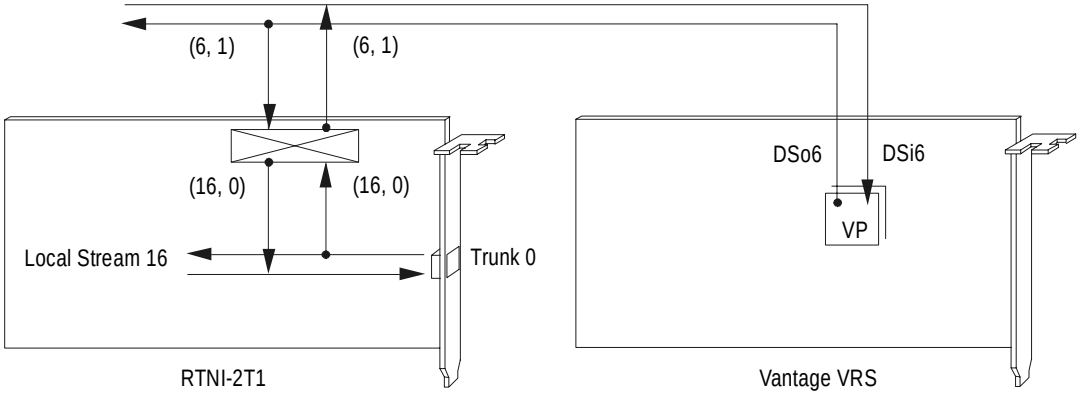

Figure 31. A Call Switched to VP Resources

Make the connection shown in Figure 31 as follows:

```
SET_OUTPUT          (INPUT {16,0}
                    OUTPUT {6,1})
SET_OUTPUT          (INPUT {6,1}
                    OUTPUT {16,0})
```

Vantage VRS, Vantage VPS, and RDSP/xx000 boards automatically have access to data that a network board places on their assigned stream. There is no need to make a connection with the resource board and the MVIP bus.

Example 9. RHT_SET_OUTPUT Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{

   BRKT_HANDLE LineHandle;              /* T1 line device handle */
   BOOLEAN IoctlResult;                 /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;           /* Bytes returned from IOCTL call
*/
   struct OutputParam_s Output;

     /* Open T1 line device */
   LineHandle = BrktOpenDevice (BRKT_DEVICE_T1_LINE, 0);

     /* Make the first unidirectional connection. */
   memset (&Output, 0, sizeof (Output));
   Output.OutputStream = 0;
   Output.OutputSlot = 1;
   Output.Mode = CONNECT_MODE;
   Output.InputStream = STREAM_MYSELF;

   IoctlResult = BrktDeviceIoControl (
         LineHandle,
         RHT_SET_OUTPUT,
         &Output,                       /* Buffer to driver */
         sizeof(Output),
         NULL,
         0,
         &BytesReturned,
         NULL);

   if (!IoctlResult)
         printf("RHT_SET_OUTPUT failed: BrktGetLastError = %d\n",
               BrktGetLastError (LineHandle));
```

Example 9. RHT_SET_OUTPUT Sample Code (Continued)

```
  /* Make the second unidirectional connection. */
   Output.OutputStream = STREAM_MYSELF;
   Output.InputStream = 0;
   Output.InputSlot = 1;

   IoctlResult = BrktDeviceIoControl (
         LineHandle,
         RHT_SET_OUTPUT,
         &Output,
         sizeof(Output),
         NULL,
         0,
         &BytesReturned,
         NULL);

   if (!IoctlResult)
         printf ("RHT_SET_OUTPUT failed: BrktGetLastError = %d\n",
               BrktGetLastError (LineHandle));

   BrktCloseDevice (LineHandle);

}
```

Connecting Line Resources

When you connect resources from two boards with switching capability in a full-duplex connection, you connect the internal resource on one board to the internal resource on another board using an intermediate MVIP stream. These types of connections are called drop and insert. You drop a call from one internal resource onto an unused MVIP timeslot, then you insert that call into an internal resource on the second board.

One example of a drop and insert connection is when you call a call center such as a mail order center. Your call reaches the call center through one of their inbound lines (through a T1 line, for example). The T1 board connects your call to a VP resource on another board. The system prompts you to enter information such as zip code, social security number, or customer number, then places your call on hold. When a customer representative is available, the T1 board switches your call to a timeslot on the MVIP bus. Another board, such as an RTNI-ATSI switches the call from the MVIP bus to an outbound line to the customer representative. In this scenario, your call is connected through the MVIP bus to an outbound line on another board.

When you perform a drop and insert, one board uses the stream in the reverse direction. For example, Figure 32 shows what would happen if two RTNI boards transmitted data on a DSix stream and received data on a DSox stream (the conventional direction). Since both boards place data on DSi0, neither receives data from the other board.

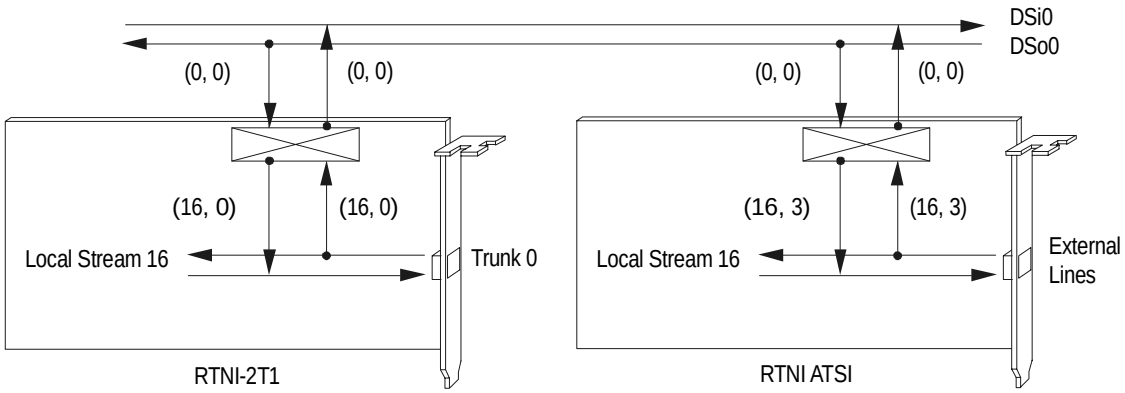

Figure 32. Two Boards Transmitting on the Same Stream

If one network board transmits on the DSix stream and receives on the DSox stream (the conventional direction), then the other network board must transmit on the DSox stream and receive on the DSix stream, as Figure 33 shows. In this configuration, the boards can connect two lines through the MVIP bus.

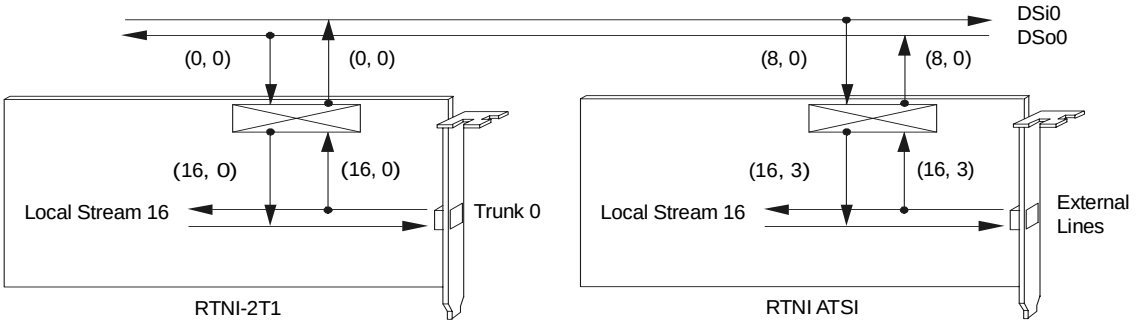

Figure 33. A Drop and Insert Connection

In Figure 33, the RTNI-ATSI board uses the streams in the reverse direction. When you call SET_OUTPUT to make a full-duplex connection between the input stream (DSi0) and the internal resource, you would use the stream reference number 8 for *InputSlot*, as Table 28 shows. The higher of the two stream numbers applies when you use a stream in the reverse direction. Again, since input and output are in reference to the resource boards, a network board using a DSi stream as input is using the stream in the reverse direction.

Make the connection shown in Figure 33 as follows:

```
ATSI board
SET_OUTPUT          (INPUT {16,3}
                    OUTPUT {8,0})
SET_OUTPUT          (INPUT {8,0}
                    OUTPUT {16,3})
T1 board
SET_OUTPUT          (INPUT {16,0}
                    OUTPUT {0,0})
SET_OUTPUT          (INPUT {0,0}
                    OUTPUT {16,0})
```

Making a Broadcast Connection

In a broadcast connection, a resource transmits data to a specified stream and timeslot, then several other resources receive data from that timeslot. In Figure 34, a board places data on stream 4, timeslot 2 (4, 2). Then several boards take data from that timeslot.

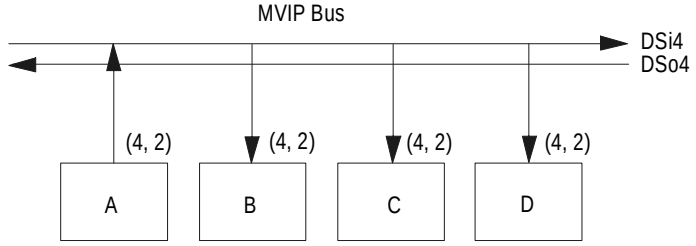

Figure 34. A Broadcast Connection and Distribution

One example of a broadcast is a hold message. One VP resource continuously plays a hold message over a specified stream and timeslot. In Figure 34, board A is transmitting the data on stream 4, timeslot 2. When calls on boards B, C, and D are put on hold, they are connected through the MVIP bus to take the hold message from stream 4 timeslot 2. Any number of resources can be connected to the broadcast timeslot at any time.

Board A can broadcast to several different boards, as shown in Figure 34. Or, board A can broadcast a call to other resources on that board.

To perform a broadcast, make a half-duplex connection between the internal resource and a timeslot on the MVIP bus. Then make a half-duplex connection between the timeslot carrying the transmitted data and the timeslots for resources to receive data.

Making Connections in an Application

An application makes and breaks connections as a call progresses. For example, the following illustrations show an incoming call to an automated attendant.

First, the T1 board receives a call, as shown in Figure 35.

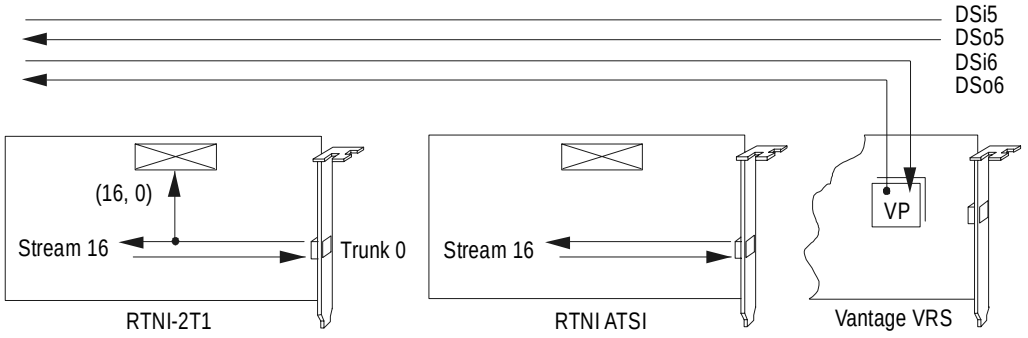

Figure 35. A T1 Board Receives a Call

Next, the T1 board connects the incoming call to a resource on a Vantage VPS board assigned to use stream 6, as shown in Figure 36. The application directs the voice resource to play a recorded message requesting an extension number.

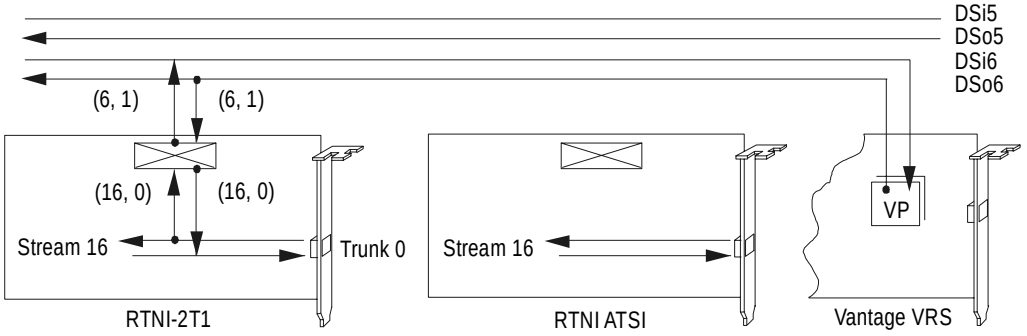

Figure 36. The T1 Transfers the Call to a VP Resource

After the caller enters an extension, the application determines which line that extension uses and makes the appropriate connection. In this example, the requested extension uses the fourth resource on the RTNI-ATSI.

The T1 board breaks the connection with the Vantage VRS and makes a connection with stream 5 on the MVIP bus. The RTNI-ATSI board also makes a full-duplex connection with stream 5. The RTNI-ATSI board then rings the extension selected by the caller.

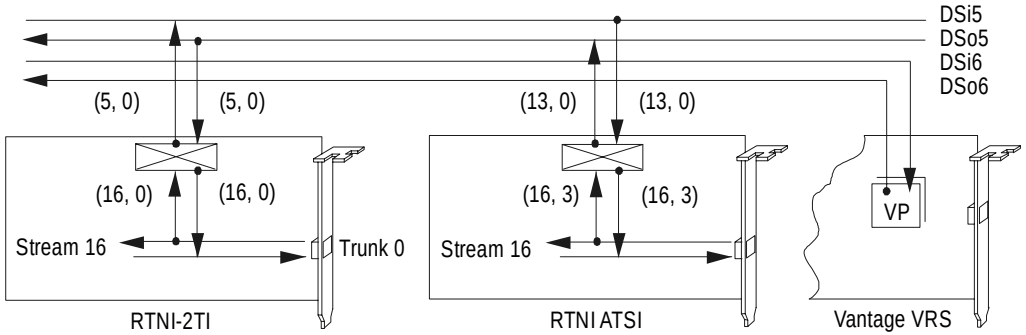

Figure 37. The T1 Transfers the Call to the Appropriate Extension

Identifying the Timeslot Mode

To find out the mode of a timeslot or connections associated with that timeslot, use the QUERY_OUTPUT function.

QUERY_OUTPUT returns the mode and connections for the specified MVIP timeslot. QUERY_OUTPUT uses *OutputParam_s.OutputStream* and *OutputParam_s.OutputSlot* to identify the timeslot. Then it populates the *OutputParam_s.Mode, OutputParam_s.InputStream, OutputParam_s.InputSlot* and *OutputParam_s.Message* fields depending on the connection mode. See the *RealCT Direct API Reference Manual* for more information about OutputParam_s.

The RHT_QRY_OUTPUT function is similar to QUERY_OUTPUT except that QUERY_OUTPUT uses a board file handle while RHT_QRY_OUTPUT uses a line file handle. Using a line file handle allows RHT_QRY_OUTPUT to use the STREAM_MYSELF value for *OutputParam_s.OutputStream*. RHT_QRY_OUTPUT is not MVIP compliant.

Example 10 shows how to use QUERY_OUTPUT to identify the timeslot connection and mode of the first T1 line.

Example 10. RHT_QRY_OUTPUT Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE LineHandle;         /* T1 line device handle */
   BOOLEAN IoctlResult;            /* Result of IOCTL call */
   BRKT_SIZE_T BytesReturned;      /* Bytes returned from IOCTL call */
   struct OutputParam_s Output;

     /* Open T1 line device */
   LineHandle = BrktOpenDevice (BRKT_DEVICE_T1_LINE, 0);
```

Example 10. RHT_QRY_OUTPUT Sample Code (Continued)

```
  /*Query unidirectional MVIP connection. Synchronous call.*/
   memset(&Output, 0, sizeof(Outout));
   Output.OutputStream = STREAM_MYSELF;

   IoctlResult = BrktDeviceIoControl (
         LineHandle,
         RHT_QRY_OUTPUT,
         &Output,                  /* Buffer to driver */
         sizeof(Output),
         &Output,                  /* Buffer to driver */
         sizeof(Output),
         &BytesReturned,
         NULL);                    /* Wait until I/O is complete */

   if (!IoctlResult)
         printf ("RHT_QRY_OUTPUT failed: BrktGetLastError = %d\n",
               BrktGetLastError (LineHandle));
   else
   {
         if (Output.Mode == CONNECT_MODE)
               printf("Connect: (%d,%d) <-- (%d,%d)\n",
                        Output.OutputStream,
                     Output.OutputSlot, Output.InputStream,
                       Output.InputSlot);
         else if (Output.Mode == PATTERN_MODE)
               printf("Pattern = %X.\n", Output.Message);
         else
               printf("Disabled (%d,%d).\n", Output.OutputStream,
                        Output.OutputSlot);
   }

   BrktCloseDevice (LineHandle);

}
```

6
MVIP-95

This chapter describes the MVIP-95 driver standard used by your H.100 bus compliant board. It includes information about the H.100 hardware standard that MVIP-95 supports.

It includes the following topics:

- Working with Computer Telephony Buses
- Defining MVIP-95
- Connecting Boards in an H.100 bus
- Mapping Board Resources
- Configuring Boards in the CT bus
- Configuring the H.100 Clock
- Configuring the H.100 Bus Speed
- Enabling or Disabling Resources
- Switching Data

Working with Computer Telephony Buses

The computer telephony (CT) bus provides a way to transfer calls between telephony resources, even if those resources are provided by different vendors. With voice, fax, video, and automatic speech recognition cards connected in a single bus, you can develop fully-integrated computer telephony applications.

A CT bus is comprised of two main parts: the physical bus and the switch block. The bus provides the physical connection between boards. The switch block provides the mechanism for switching data between telephony resources.

Only the RTNI series boards, RealBLOCs series boards, and Vantage PCI boards provide a switch block and can make physical connections between resources. RTNI series boards do not provide VP resources, but they can switch data to VP resources on other boards. The Vantage VPS, Vantage VRS, and RDSP/xx000 boards do not provide switching capability. They provide VP resources over the CT bus. The following table outlines board properties:

Board Family	MVIP 90	MVIP 95	MVIP 90 bus	H.100 bus
RDSP	x		x	
RTNI	x		x	
Vantage PCI	x	x	x	x
RealBLOCs PCI	x	x	x	x

Brooktrout supports two levels of MVIP:

- RDSP, Vantage VRS, Vantage VPS, and RTNI boards all use the MVIP-90 software and hardware standard.
- The Vantage PCI and RealBLOCs products support the H.100 Computer Telephony (CT) bus. H.100 is a hardware standard. The MVIP-95 software standard supports the H.100, MVIP-90, and H-MVIP buses.

The *RealCT Direct API Reference Manual* has functions for both the MVIP-90 and MVIP-95 standards. Be sure you use the functions appropriate for the boards in your system.

- Use MVIP-90 functions for all boards if they are connected in an MVIP-90 bus. For more information about the MVIP-90 standard, see Chapter 5, *MVIP-90,* on page 143.
- Use MVIP-95 functions for boards connected in an H.100 bus.

Defining MVIP-95

MVIP-95 is a software device driver standard that supports the MVIP-90, and H.100 hardware standards.

You can connect an H.100 compliant board to RTNI or Vantage VRS and VPS products using the MVIP bus, or to boards that support H.100 using an H.100 bus.

If you have several H.100 bus compliant PCI boards connected in an H.100 bus, use MVIP-95 functions for all boards in the system. If you have an H.100 bus compliant PCI board connected to MVIP-90 compliant boards through an MVIP bus, you would use MVIP-90 functions for all boards, including the H.100 bus compliant board.

MVIP-95 has the following changes from MVIP-90:

- Supports H.100
- Removes directionality from stream numbers
- Changes the numbering convention for streams
- Separates CT bus clock and SEC8K configuration

The RealCT Direct API includes 14 functions for MVIP-95 switching as shown in Table 29.

Table 29. Functions Specific to Either MVIP-90 or MVIP-95

MVIP-90 Functions	MVIP-95 Functions
RESET_SWITCH	MVIP95_CMD_RESET_SWITCH
SAMPLE_INPUT	MVIP95_CMD_SAMPLE_INPUT
SET_OUTPUT	MVIP95_CMD_SET_OUTPUT
CONFIG_CLOCK	MVIP95_CMD_CONFIG_BOARD_CLOCK
CONFIG_CLOCK	MVIP95_CMD_CONFIG_SEC8K_CLOCK
no MVIP-90 equivalent	MVIP95_CMD_CONFIG_NETREF_CLOCK
no MVIP-90 equivalent	MVIP95_CMD_CONFIG_STREAM_SPEED
QUERY_OUTPUT	MVIP95_CMD_QUERY_OUTPUT
no MVIP-90 equivalent	MVIP95_CMD_QUERY_BOARD_INFO
no MVIP-90 equivalent	MVIP95_CMD_QUERY_DRIVER_INFO
no MVIP-90 equivalent	MVIP95_CMD_QUERY_BOARD_CLOCK
no MVIP-90 equivalent	MVIP95_CMD_QUERY_STREAM_SPEED
QUERY_SWITCH_CAPS	MVIP95_CMD_QUERY_SWITCH_CAPS
TRISTATE_SWITCH	MVIP95_CMD_SET SWITCH
DUMP_SWITCH	DUMP_SWITCH

Understanding H.100 Architecture

H.100 is a high-speed hardware standard. Previous to H.100 there were two telephony bus standards: MVIP and SCSA. Boards with an MVIP bus could not communicate with boards using a SCSA bus. Boards using either the MVIP or SCSA standard can communicate through the H.100 bus, however, allowing developers to have boards from different vendors in a single bus.

The H.100 bus supports 32 8.192-Mb/s streams, as shown in Figure 38. Each H.100 stream has a maximum of 128 64-Kb/s time division multiplexed slots, each of which supports a single resource. (128 slots x 64 Kb/s/slot = 8.192 Mb/s). Each slot within a specific stream is called a timeslot. Each timeslot on the H.100 bus can carry voice data between resources.

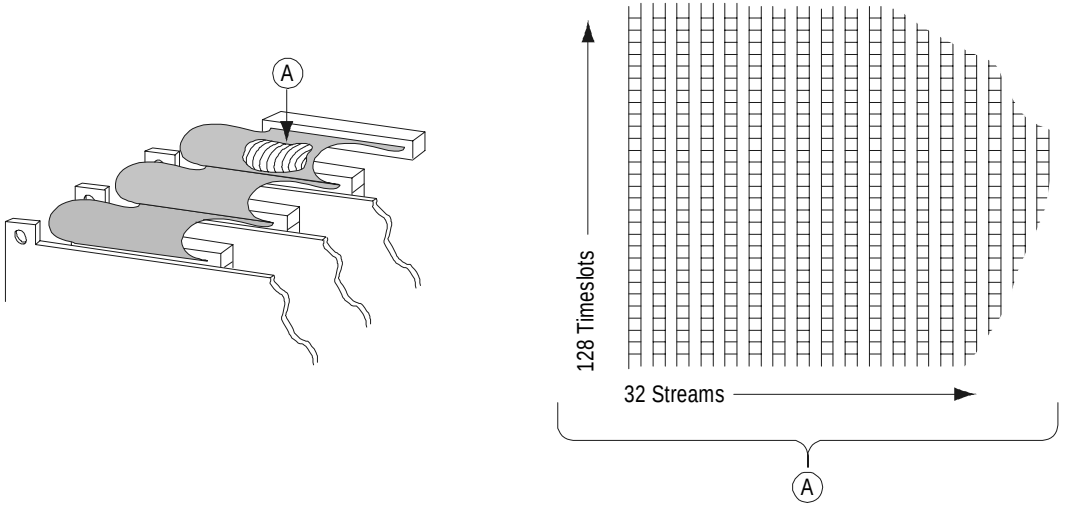

Figure 38. Data Streams in an H.100 Bus

When you adjust the stream speed, you change the number of timeslots the stream allocates. This provides compatibility with the MVIP bus, which only provides 32 timeslots per stream. For more information about how to adjust the stream speed, see *Configuring the H.100 Bus Speed* on page 204.

Data in individual streams are formatted into frames. Each frame contains 8 bits of information for each of the timeslots, as shown in Figure 39. Each 8-bit timeslot has a period of 125 µs, for a total of 8000 timeslots per second (8 bits/slot x 8000 slots/second = 64 Kb/s).

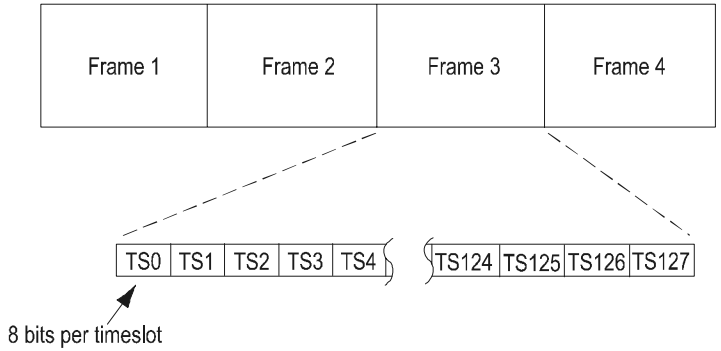

Figure 39. Timeslots in an H.100 Frame

Connecting Boards in an H.100 bus

This section provides information about the H.100 bus, including information about how to set clocks and stream speeds for boards connected in an H.100 bus.

This section only applies to boards that are physically connected in an H.100 bus to other H.100-compliant boards. For information about connecting your H.100-compliant board in an MVIP bus, see *Connecting Boards in an MVIP-90 Bus* on page 212.

Numbering Streams

The MVIP-95 standard defines CT bus and local streams.

- CT bus streams are streams that connect the boards in a CT bus. These are the streams running through the physical H.100 connector.
- Local streams carry the board's internal line and VP resources.

Both sets of streams are numbered sequentially, starting with 0.

Streams in an H.100 bus are not directional; boards can transmit or receive data on any stream. This lack of directionality in H.100 streams is in contrast to MVIP-90 streams, where the streams are divided into DSix (input) and DSox (output) directions, as described in *Working with MVIP-90 Data Streams* on page 145. However, when a board that uses the H.100 standard is connected through an MVIP bus, the application adheres to the MVIP stream directionality.

Mapping Board Resources

The MVIP-95 board uses four internal data streams for a four-port board, eight internal data streams for an eight-port board, or 12 internal data streams for a 12-port board. Each stream carries four resources and uses four timeslots on the stream. The stream and timeslot assignments are set by the hardware and are not configurable.

In the case of the Vantage PCI board, the first four VP resources use streams 0 and 1 to transmit and receive data. The first four line resources use streams 2 and 3 to transmit and receive data. The second four VP resources use streams 4 and 5 to transmit and receive data. The second four line resources use streams 6 and 7 to transmit and receive data. Figure 40 shows local streams numbered on a Vantage PCI board.

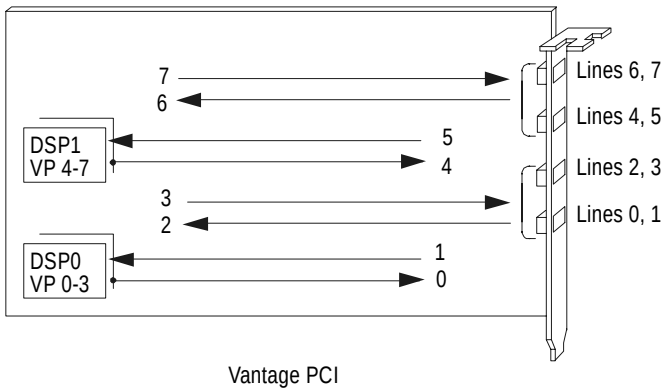

Figure 40. Local Streams on a Vantage PCI Board

The resources on each stream are assigned sequentially to timeslots 0, 8, 16, and 24, as shown in Table 30. For example, the first VP resource would transmit data on input stream 0, timeslot 0. It would receive data on output stream 1, timeslot 0. The second VP resource would receive data on stream 0, timeslot 8.

As Table 30 shows, the streams are designated input and output. Those designations are in reference to the switch block. A resource transmits data on the input stream (input to the switch block) and receives data on the output stream (output from the switch block).

Table 30. Stream Assignments for Vantage PCI Boards

Stream number	Assignment	Resource	Timeslot
0	Input	VP0	0
		VP1	8
		VP2	16
		VP3	24
1	Output	VP0	0
		VP1	8
		VP2	16
		VP3	24
2	Input	Line0	0
		Line1	8
		Line2	16
		Line3	24
3	Output	Line0	0
		Line1	8
		Line2	16
		Line3	24
4	Input	VP4	0
		VP5	8
		VP6	16
		VP7	24
5	Output	VP4	0
		VP5	8
		VP6	16
		VP7	24

Stream number	Assignment	Resource	Timeslot
6	Input	Line4	0
		Line5	8
		Line6	16
		Line7	24
7	Output	Line4	0
		Line5	8
		Line6	16
		Line7	24

Mapping RealBLOCs Resources

This section lists the MVIP-95 local stream assignments.

Table 31. RealBLOCs MVIP-95 Local Stream Assignments

Stream number	Assignment	Resource	Timeslot
0	Input	Line0	0
		Line1	8
		Line2	16
		Line3	24
1	Output	Line0	0
		Line1	8
		Line2	16
		Line3	24

Stream number	Assignment	Resource	Timeslot
2	Input	Line4	0
		Line5	8
		Line6	16
		Line7	24
3	Output	Line4	0
		Line5	8
		Line6	16
		Line7	24
4	Input	Line8	0
		Line9	8
		Line10	16
		Line11	24
5	Output	Line8	0
		Line9	8
		Line10	16
		Line11	24
6	Input	Line12	0
		Line13	8
		Line14	16
		Line15	24
7	Output	Line12	0
		Line13	8
		Line14	16
		Line15	24

Stream number	Assignment	Resource	Timeslot
8	Input	VP16	0
		VP17	8
		VP18	16
		VP19	24
9	Output	Line16	0
		Line17	8
		Line18	16
		Line19	24
10	Input	Line20	0
		Line21	8
		Line22	16
		Line23	24
11	Output	Line20	0
		Line21	8
		Line22	16
		Line23	24

Configuring Boards in the CT bus

There are several steps to configuring your CT bus. The following sections in this chapter discuss these steps in more detail.

1. Enable or disable resources.

   By default, the Vantage PCI switch block is not available to the CT bus. Enable the switch block before using the switch block. The RealBLOCs board does not provide the switch enable option. The switch is always enabled.

2. Configure the CT bus clock.

   The CT bus uses a clock to synchronize frames. You set which board drives the clock for the system.

3. Configure the CT bus speed.

   The H.100 streams can operate at different speeds. Select a bus speed that is compatible with the other boards in the system.

4. Establish connections.

   When you switch calls, you establish connections between a board's resources and the CT bus.

Enabling or Disabling Resources

By default, the entire Vantage PCI board switch block is not available to the CT bus. To enable or disable the MVIP-95 switch block, use the MVIP95_CMD_SET_SWITCH function. The RealBLOCs board does not provide the switch enable option. The switch is always enabled.

In their default state, lines and VP resources on a Vantage PCI board are combined to make channels. Enabling the switch block on a Vantage PCI board makes the VP and LS resources available to the bus.

Enable the Vantage PCI switch block before you configure the clock, set the stream speed, or establish connections for that board. Example 11 shows how to enable the switch block on a Vantage PCI board.

Example 11. MVIP95_CMD_SET_SWITCH Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE BoardHandle;
   BOOLEAN IoctlResult;
   BRKT_SIZE_T BytesReturned;
   struct MVIP95_SET_SWITCH_PARMS SwitchParams;

   BoardHandle = BrktOpenDevice(BRKT_DEVICE_RDSP_BOARD, 0);
   if (BoardHandle == BRKT_INVALID_HANDLE_VALUE)
         return;

   memset(&SwitchParams, 0, sizeof(SwitchParams));
   SwitchParams.bus_enable_state = TRUE;
```

Example 11. MVIP95_CMD_SET_SWITCH Sample Code (Continued)

```
   IoctlResult = BrktDeviceIoControl(
         BoardHandle,
         MVIP95_CMD_SET_SWITCH,
         &SwitchParams,
         sizeof(SwitchParams),
         NULL,
         0,
         &returned,
         NULL);


   if (!IoctlResult)
         printf("MVIP95_CMD_SET_SWITCH failed:"
               "BrktGetLastError = %d\n",
               BrktGetLastError (BoardHandle));


   BrktCloseDevice (BoardHandle);


}   /* main */
```

Configuring the H.100 Clock

You can connect an H.100 compliant board to either the MVIP or H.100 bus. You set the clocks differently depending on which bus your board is connected to.

Boards connected in an H.100 bus use two primary clocks to synchronize frames. These are the A clock and the B clock. The H.100 bus also provides NETREF to synchronize boards with a digital interface to the network. Since the H.100 compliant boards do not have a digital interface, you can not set the NETREF clock for those boards.

In an H.100 bus, one board drives the A clock, another board drives the B clock, and all other boards take the clock from either A or B.

Setting the Primary Clock

Set the H.100 clock during driver configuration using the Configuration Wizard. For specific information about how to set the H.100 clock during driver configuration, see the installation and configuration guide that came with your software.

If you need to change clock parameters after installation, use the MVIP95_CMD_CONFIG_BOARD_CLOCK function. If your board is connected to an H.100 bus, use the *H.100* data structure to set the primary clock.

In the *H.100* data structure you specify whether the board sets the A clock, sets the B clock, or receives the clock signal from either the A or B primary clock. In most systems, it is best to have one board drive the A clock, one board drive the B clock, and all other boards use the A clock as a reference.

If the A clock fails, all boards in the system switch to using the B clock as long as the boards have fallback enabled. The clocks stay in this mode until reprogrammed by the application.

For example, if you have a system with three H.100 compliant boards, set the clock as follows:

1. Set the first board to drive the A clock (MVIP95_H100_MASTER_A) and to drive the clock from the internal oscillator (MVIP95_SOURCE_INTERNAL)
2. Set the second board to drive the B clock (MVIP95_H100_MASTER_B) and to drive the clock from the internal oscillator (MVIP95_SOURCE_INTERNAL)
3. Set the third board to receive the clock from the A clock on the H.100 bus ((MVIP95_H100_SLAVE), (MVIP95_SOURCE_H100_A) .

Example 12 shows how to set the H.100_A clock from its internal oscillator.

Example 12. MVIP95_CMD_CONFIG_BOARD_CLOCK (H.100) Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE BoardHandle;
   BOOLEAN IoctlResult;
   BRKT_SIZE_T BytesReturned;
   struct MVIP95_CONFIG_H100_BOARD_CLOCK_PARMS Param;

   BoardHandle = BrktOpenDevice(BRKT_DEVICE_RDSP_BOARD, 0);
   if (BoardHandle == BRKT_INVALID_HANDLE_VALUE)
         return;

   memset (&Param, 0, sizeof(Param));
   Param.size = sizeof(Param);
   Param.clock_type = MVIP95_H100_CLOCKING;
   Param.clock_source = MVIP95_SOURCE_INTERNAL;
   Param.h100_clock_mode = MVIP95_H100_MASTER_A;
   Param.auto_fall_back = MVIP95_H100_DISABLE_AUTO_FB;

     /* Set H100 clock */
   IoctlResult = BrktDeviceIoControl(
         BoardHandle,
         MVIP95_CMD_CONFIG_BOARD_CLOCK,
         &Param,
         sizeof(Param),
         NULL,
         0,
         &BytesReturned,
         NULL);
```

Example 12. MVIP95_CMD_CONFIG_BOARD_CLOCK (H.100) Sample Code (Continued)

```
   if (!IoctlResult)
          printf("MVIP95_CMD_CONFIG_BOARD_CLOCK failed:"
                 "BrktGetLastError = %d\n",
                 BrktGetLastError (BoardHandle));


   BrktCloseDevice (BoardHandle);


}  /* main */
```

Setting the NETREF Clock

The NETREF clock synchronizes the CT bus clock with a digital network. Since an H.100 compliant board does not have a digital network, you cannot set the NETREF clock for that board.

If you have a board with a digital trunk in your system, use the MVIP95_CMD_CONFIG_NETREF_CLOCK function to determine which board sets the NETREF clock.

Configuring the H.100 Bus Speed

The H.100 hardware specification supports three possible bus speeds. These speeds control how many timeslots the streams use as follows:

- If the streams use all 128 timeslots, each carrying data at 64 Kb/s, the stream operates at 128 x 64 Kb/s = 8.192 Mb/s.
- If the streams use 64 timeslots, each carrying data at 64 Kb/s, the stream operates at 64 x 64 Kb/s = 4.096 Mb/s.
- If the streams use 32 timeslots, each carrying data at 64 Kb/s, the stream operates at 32 x 64 Kb/s = 2.048 Mb/s.

These different stream speeds provide compatibility with MVIP-90, H-MVIP, and SCSA buses.

To set the bus speed, use the MVIP95_CMD_CONFIG_STREAM_SPEED function. You set the stream speed for the first 16 streams in groups of four, so you would set the speed for streams 0-3 together; 4-8 together; 9-12 together; and 13-16 together. You can choose between a stream speed of 8 (8.192 MHz), 4 (4.096 MHz), or 2 (2.048 MHz). Streams 0-15 have a default speed of 2.048 MHz.

The last 16 streams have an undefined stream speed. They are not available unless you assign a stream speed.

If you have several H.100 compliant boards connected in an H.100 bus, all boards should be using a bus speed of 8 to take full advantage of the H.100 bus. However, if you do adjust the speed for one board, adjust all boards so they are using the same speed.

To find out what bus speed the CT bus is using, use the MVIP95_CMD_QUERY_STREAM_SPEED function.

Example 13 shows how to configure the streams to use the fastest stream speed.

Example 13. MVIP95_CMD_CONFIG_STREAM_SPEED Sample Code

```
#include “brktddm.h”

int main(int argc, char **argv)
{
   BRKT_HANDLE BoardHandle;
   BOOLEAN IoctlResult;
   BRKT_SIZE_T BytesReturned;
   struct MVIP95_CONFIG_STREAM_SPEED_PARMS Speed;

   BoardHandle = BrktOpenDevice(BRKT_DEVICE_RDSP_BOARD, 0);
   if (BoardHandle == BRKT_INVALID_HANDLE_VALUE)
         return;

   memset(&Speed, 0, sizeof(Speed));
   Speed.size = sizeof(Speed);
   Speed.speed = MVIP95_8MBPS_STREAM_SPEED;
   Speed.stream[ 0 ] = 0;

   /* Configure the speed of a group of streams on the CT bus */
   IoctlResult = BrktDeviceIoControl(
         BoardHandle,
         MVIP95_CMD_CONFIG_STREAM_SPEED,
         &Speed,                  /* Buffer to driver */
         sizeof(Speed),           /* Length of buffer */
         NULL,
         0,
         &BytesReturned,
         NULL);                   /* Wait until I/O complete */

   if (!IoctlResult)
         printf("MVIP95_CMD_CONFIG_STREAM_SPEED failed:"
               "BrktGetLastError = %d\n",
               BrktGetLastError (BoardHandle));

   BrktCloseDevice (BoardHandle);
}  /* main */
```

Switching Data

A board with a switch block can transfer data between resources on any board connected through the CT bus. A board such as the RTNI series, which only provides line resources, would switch a call to a Vantage VRS to process the call. Only the RTNI, RealBLOCs, and Vantage PCI series boards provide switch blocks. Although Vantage VRS, Vantage VPS, and RDSP/xx000 boards connect to the CT bus, they do not have a switch block and can only process data switched to their resources by another board.

Throughout this section, boards connected in a CT bus are depicted as shown in Figure 41.

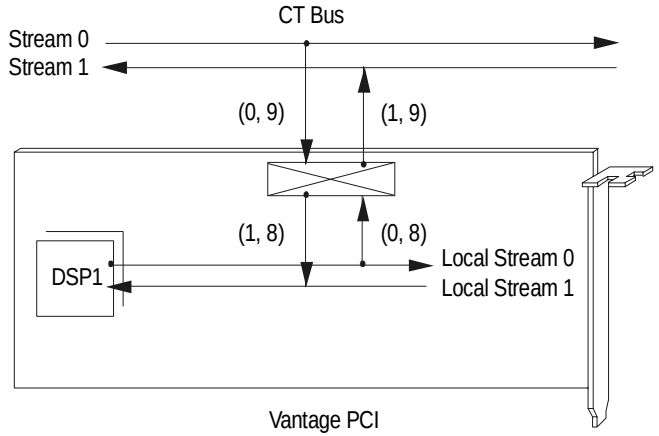

Figure 41. A Board in a CT bus

In Figure 41, the arrows above the boards represent streams in the CT bus. Streams running within the board to and from internal line or VP resources represent local streams. The rectangle with an X is the switch block. Arrows connecting CT bus and local resources through the switch block show which streams the switch block is switching data between. Numbers beside the arrows define the stream and timeslot that the switch block is switching data between.

Using MVIP-95 Switching Functions

When you switch calls using the MVIP-95 switch block, use the MVIP95_CMD_SET_OUTPUT function. Unlike SET_OUTPUT in MVIP-90, MVIP95_CMD_SET_OUTPUT requires that you specify CT_Bus or local stream, since stream numbering for both begins with 0.

For example, Figure 42 shows a full duplex connection between the second VP resource and the CT bus.

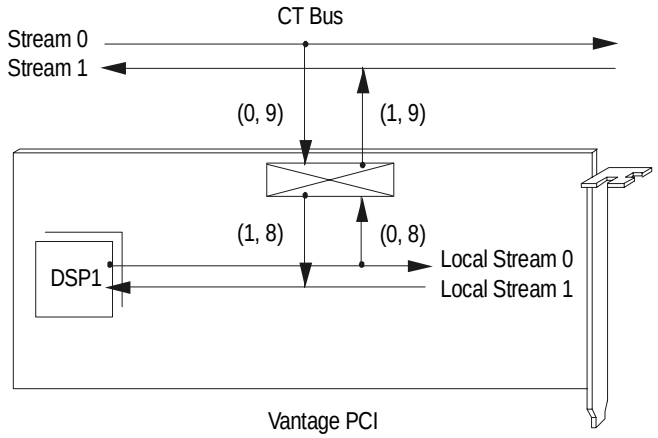

Figure 42. A Full-duplex Connection With the Vantage PCI

Make the full-duplex connection shown in Figure 42 as follows:

```
MVIP95_CMD_SET_OUTPUT     (input {LOCAL,0,8}
                          output {CT_Bus,1,9})
MVIP95_CMD_SET_OUTPUT     (input {CT_Bus,0,9}
                          output {LOCAL,1,8})
```

Establishing Connections

Connections Between Local Resources

An H.100 compliant board can establish connections between local resources. For example, if a call comes in on the first line, the switch block can connect that call to any VP resource. If the switch block is enabled, the first line resource is not automatically connected to the first VP resource to form a channel.

In Figure 43, a Vantage PCI board switch block has established a full-duplex connection between the first incoming line and the third internal VP resource, forming a channel.

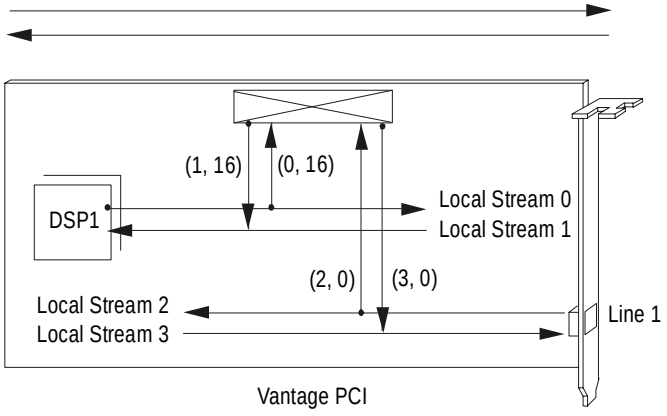

Figure 43. A Connection Between Local Resources

Make the full-duplex connection shown in Figure 43 as follows:

```
MVIP95_CMD_SET_OUTPUT     (input {LOCAL,0,16}
                          output {LOCAL,3,0})
MVIP95_CMD_SET_OUTPUT     (input {LOCAL,2,0}
                          output {LOCAL,1,16})
```

Connections Through the CT bus

See *Switching Calls through the MVIP-90 Bus* on page 165 for a description of connections that you can establish through a CT bus, such as drop and insert and broadcast connections. The concepts described in that section apply to switching calls using the H.100 bus. However, in H.100 there is no need to worry about stream direction.

Example 14 shows how to use the MVIP95_CMD_SET_OUTPUT function to make a full-duplex connection with the first line resource on a Vantage PCI and the first and second CT_bus stream.

Example 14. MVIP95_CMD_SET_OUTPUT Sample Code

```
#include “brktddm.h”

/* The following setup allows 2 connections to be made with
   one device driver request */

typedef struct
{
   struct MVIP95_SET_OUTPUT_PARMS Parms;
   struct MVIP95_OUTDESC AdditionalOutputs[1];

} SET_OUTPUT_PARMS;

int main(int argc, char **argv)
{
   BRKT_HANDLE BoardHandle;
   BOOLEAN IoctlResult;
   BRKT_SIZE_T BytesReturned;
   struct MVIP95_OUTDESC *pOutput;
   struct SET_OUTPUT_PARMS Connect;

   BoardHandle = BrktOpenDevice(BRKT_DEVICE_RDSP_BOARD, 0);
   if (BoardHandle == BRKT_INVALID_HANDLE_VALUE)
          return;
```

Example 14. MVIP95_CMD_SET_OUTPUT Sample Code (Continued)

```
memset(&Connect, 0, sizeof(Connect));
Connect.parms.size = sizeof(Connect);

/* Connect { CT_BUS, 1, 1 } <= { LOCAL, 2, 0 } */
pOutput = &(Connect.parms.output[0]);
pOutput->mode = MVIP95_CONNECT_MODE;

/* Output stream / timeslot */
pOutput->terminus.bus = MVIP95_CT_BUS;
pOutput->terminus.stream = 1;
pOutput->terminus.timeslot = 1;

/* Input stream / timeslot */
pOutput->connected_from.bus = MVIP95_LOCAL_BUS;
pOutput->connected_from.stream = 2;
pOutput->connected_from.timeslot = 0;

/* Connect { LOCAL, 3, 0 } <= { CT_BUS, 0, 1 } */
pOutput = &(Connect.parms.output[1]);
pOutput->mode = MVIP95_CONNECT_MODE;

/* Output stream / timeslot */
pOutput->terminus.bus = MVIP95_LOCAL_BUS;
pOutput->terminus.stream = 3;
pOutput->terminus.timeslot = 0;

/* Input stream / timeslot */
pOutput->connected_from.bus = MVIP95_CT_BUS;
pOutput->connected_from.stream = 0;
pOutput->connected_from.timeslot = 1;
```

Example 14. MVIP95_CMD_SET_OUTPUT Sample Code (Continued)

```
   /* Perform a FULL-DUPLEX connection */
   IoctlResult = BrktDeviceIoControl(
          BoardHandle,
          MVIP95_CMD_SET_OUTPUT,
          &Connect,
          sizeof(Connect),
          NULL,
          0,
          &BytesReturned,
          NULL);


   if (!IoctlResult)
          printf("MVIP95_CMD_SET_OUTPUT failed:"
                 "BrktGetLastError = %d\n",
                 BrktGetLastError (BoardHandle));


   BrktCloseDevice (BoardHandle);


}   /* main */
```

Connecting Boards in an MVIP-90 Bus

This section provides information about connecting an H.100 compliant board in an MVIP-90 bus. When you connect an H.100 compliant board to boards in an MVIP-90 bus, you use different bus speeds, clock settings, stream numbering, and switching functions than if the board is connected in an H.100 bus.

You also use the MVIP-90 functions when an H.100 compliant board is connected in an MVIP-90 bus. Use the following MVIP-90 functions with an H.100 compliant board:

- RESET_SWITCH
- QUERY_SWITCH_CAPS
- SET_OUTPUT
- QUERY_OUTPUT
- SAMPLE_INPUT
- CONFIG_CLOCK
- TRISTATE_SWITCH

Before reading this section, be sure you are familiar with the information and function examples in Chapter 5, *MVIP-90,* on page 143.

Mapping Resources for MVIP-90

When you use MVIP-90 functions with an H.100 compliant board, you also use equivalent MVIP-90 local stream numbering. Table 32 shows the relationship between MVIP-90 and MVIP-95 local stream numbering on the Vantage PCI board. Table 32 shows the MVIP-95 local stream numbering on the RealBLOCs board.

Vantage PCI resources

As Table 32 shows, all VP resources use stream 16 and all line resources use stream 18. Resources are assigned to timeslots sequentially starting with 0. For example, the eight VP resources use timeslots 0 through 7 on stream 16. The eight line resources use timeslots 0 through 7 on stream 18.

Table 32. Comparing Vantage PCI Internal Stream Numbering

Resource	MVIP-90 Numbering		MVIP-95 Numbering		
	Stream	Timeslot	Streams		Timeslot
			input	output	
VP0	16	0	0	1	0
VP1	16	1	0	1	8
VP2	16	2	0	1	16
VP3	16	3	0	1	24
VP4	16	4	4	5	0
VP5	16	5	4	5	8
VP6	16	6	4	5	16
VP7	16	7	4	5	24
LS0	18	0	2	3	0
LS1	18	1	2	3	8
LS2	18	2	2	3	16
LS3	18	3	2	3	24
LS4	18	4	6	7	0
LS5	18	5	6	7	8
LS6	18	6	6	7	16
LS7	18	7	6	7	24

Figure 44 shows the internal stream numbering for a Vantage PCI board.

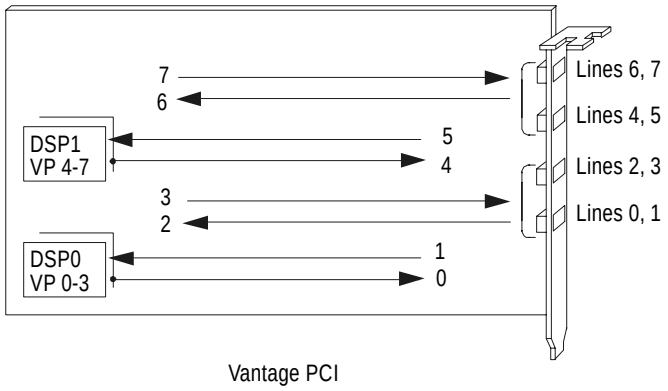

Figure 44. Vantage PCI internal Stream Numbering in MVIP-90

RealBLOCs resources Table 33 lists the MVIP-90 local stream assignments.

Table 33. RealBLOCs Local MVIP-90 Stream Assignments

Resource	MVIP-90 Numbering		MVIP-95 Numbering		
	Stream	Timeslot	Streams		Timeslot
			input	output	
Line 0	16	0	16	0	0
Line 1	16	1	16	1	8
Line 2	16	2	16	2	16
Line 3	16	3	16	3	24
Line 4	16	4	16	4	0
Line 5	16	5	16	5	8
Line 6	16	6	16	6	16
Line 7	16	7	16	7	24
Line 8	18	8	18	8	0
Line 9	18	9	18	9	8
Line 10	18	10	18	10	16
Line 11	18	11	18	11	24
Line 12	16	12	16	12	16
Line 13	16	13	16	13	24
Line 14	18	14	18	14	0
Line 15	18	15	18	15	8
Line 16	18	16	18	16	16
Line 17	16	17	16	17	16
Line 18	16	18	16	18	24
Line 19	18	19	18	19	0
Line 20	18	20	18	20	8
Line 21	18	21	18	21	16
Line 22	18	22	18	22	24
Line 23	18	23	18	23	8

Configuring the H.100 Clock for the MVIP-90 Bus

Defining H.100 Compatibility Clocks

Boards connected in an MVIP bus use an 8-kHz clock signal to synchronize frames. One H.100 compliant board in the bus drives the clock. All other boards receive their clock signal through the 8-kHz clock signal set by the H.100 compliant board.

The H.100 provides clocks that are compatible with the MVIP-90 clocks. These are shown in Table 34:

Table 34. H.100 Compatibility Clocks

H.100 clock	MVIP-90 Clock
/FR_COMP	/F0
/C4	/C4
C2	C2
CT_NETREF	SEC8K

If you have an T1 or E1 board in the system, that board should drive the /F0 clock. These boards get timing information for T1 and E1 signals from the network. All other boards in the system should receive the clock signal from the MVIP bus. If there are no T1 or E1 boards in the system, an H.100 compliant board can use the internal oscillator to drive the clock. Vantage VRS, Vantage VPS, and RDSP/xx000 boards can not set the MVIP clock. They must receive the clock from the MVIP bus.

You can also specify that boards use SEC8K as a backup signal. SEC8K is used as a fallback if the primary signal fails. As with the primary signal, only one board drives the SEC8K clock.

Setting the Clock Parameters

Configure the clock parameters using the Configuration Wizard during software installation. For specific information about the Configuration Wizard, see the installation and configuration guide that came with your software.

If you want to change your configuration after software installation, use the MVIP-90 function CONFIG_CLOCK, as described in *Configuring the MVIP-90 Clock* on page 150. You would set the clock for the Vantage PCI board similar to an RTNI-ATSI or RealBLOCs board: It should only drive the clock if there is no T1 or E1 in the system and it drives the clock from its internal oscillator.

You can also use the MVIP95_CMD_CONFIG_BOARD_CLOCK MVIP-95 function to set the clock. Use the MVIP data structure to set the compatibility clocks for MVIP-90. We do not recommend using the MVIP-95 function if your H.100 compliant board is connected in an MVIP-90 bus.

Configuring the H.100 Stream Speed for the MVIP-90 Bus

The MVIP-90 standard only supports 32 timeslots and 16 streams, so an H.100 compliant board connected in an MVIP-90 bus can also only use 32 timeslots, or a speed of 2 MHz, for its first 16 streams.

For example, if you have a Vantage PCI or RealBLOCs board and an RTNI-ATSI connected in an MVIP-90 bus, set the first 16 streams of the Vantage PCI or RealBLOCs board to use a bus speed of 2 MHz, which is the default value for those streams.

For an example of how to configure stream speeds, see Example 13 on page 205.

Switching Data

If a Vantage PCI or RealBLOCs board is connected in an MVIP bus, you adhere to the stream directionality of the MVIP bus. We recommend that you use the MVIP-90 function SET_OUTPUT to switch data over the MVIP-90 bus, but you can also use the MVIP-95 functions for the Vantage PCI or RealBLOCs board. You would use the MVIP-90 functions for any MVIP-90 board in the system.

Using MVIP-90 functions

When you connect a Vantage PCI or RealBLOCs board to the MVIP-90 bus, use the MVIP-90 function SET_OUTPUT and the MVIP-90 stream directionality and stream numbering. The Vantage PCI or RealBLOCs board functions like an RTNI board with VP resources; it uses internal streams 16 and 18, and uses DSix and DS0x streams to switch data over the CT bus.

All the concepts described in *Switching Calls through the MVIP-90 Bus* on page 165 apply to the Vantage PCI or RealBLOCs board when it is connected in an MVIP-90 bus. This includes stream numbering and how connections are made. When you switch data using MVIP-90 functions, you use the internal stream 16 and 18 as described in *Mapping Resources for MVIP-90* on page 212 or, for the RealBLOCs board, see Table 33 on page 215.

Using MVIP-95 functions

We do not recommend that you use MVIP-95 functions when you connect the Vantage PCI or RealBLOCs board to an MVIP-90 bus. However, if you do choose to use the MVIP95_CMD_SET_OUTPUT function, use MVIP-95 stream numbering when you switch data.

Although MVIP-90 and MVIP-95 number streams differently, the streams are physically connected. Table 35 shows the relationship between MVIP-90 and MVIP-95 stream numbering

Table 35. Relationship Between MVIP-90 and MVIP-95 Stream Numbering.

MVIP-90	MVIP-95
DSo0	0
DSi0	1
DSo1	2
DSi1	3
DSo2	4
DSi2	5
DSo3	6
DSi3	7
DSo4	8
DSi4	9
DSo5	10
DSi5	11
DSo6	12
DSi6	13
DSo7	14
DSi7	15

For example, if a Vantage VRS or RealBLOCs board is using stream 6, transmit a call to that board from a Vantage PCI using the MVIP-95 equivalent to DSi6, which is stream 13.

Figure 45 shows an H.100 compliant board connecting a call to a Vantage VRS that is assigned to use stream 6. The MVIP-95 function to switch the call uses stream numbers 13 and 12, which correspond to DSi6 and DSo6.

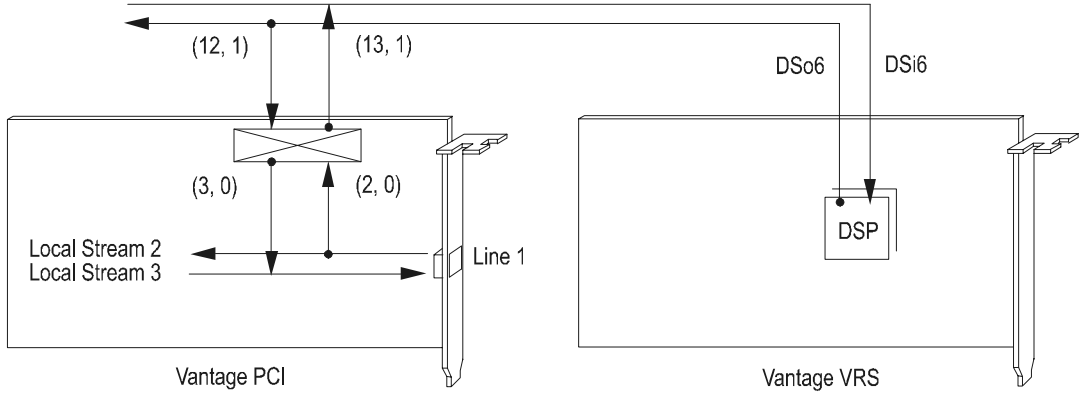

Figure 45. A Vantage PCI Transferring a Call to a Vantage VRS

Make the full-duplex connection shown in Figure 45 as follows:

```
MVIP95_CMD_SET_OUTPUT     (input {LOCAL,2,0}
                          output {CT_Bus,13,1})
MVIP95_CMD_SET_OUTPUT     (input {CT_Bus,12,1}
                          output {LOCAL,3,0})
```

Figure 46 shows a connection between a Vantage PCI and an RTNI-ATSI. In this connection, both boards have a switch block. You would use the MVIP-95 function and MVIP-95 stream numbering to make the full-duplex connection between the Vantage PCI board's internal resources and the MVIP stream. Then you would use the MVIP-90 functions and MVIP-90 stream numbering to make a full-duplex connection between the RTNI board's internal resources and the MVIP stream.

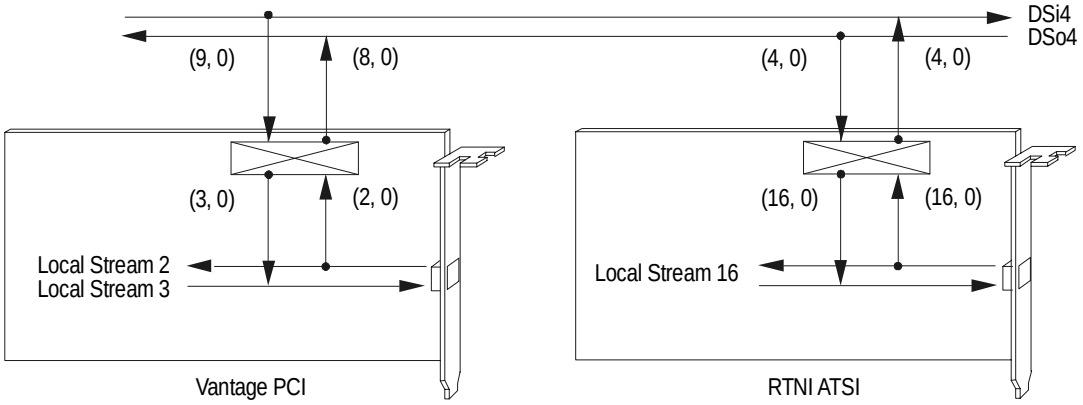

Figure 46. A Drop and Insert Connection With a Vantage PCI

Make the full-duplex connection shown in Figure 46 as follows:

```
MVIP95_CMD_SET_OUTPUT     (input {LOCAL,2,0}
                          output {CT_Bus,8,0})
MVIP95_CMD_SET_OUTPUT     (input {CT_Bus,9,0}
                          output {LOCAL,3,0})
SET_OUTPUT                (input {4,0}
                          output {16,0})
SET_OUTPUT                (input {16,0}
                          output {4,0})
```

Appendix A
T1 Line Protocols

This appendix describes the signaling bits transmitted in the most common T1 line protocols.

It includes the following sections:

- Overview of Protocols
- Immediate Start
- Wink Start
- Double Wink Start
- Loop Start
- Ground Start

Overview of Protocols

The Immediate Start, Wink Start, and Double Wink Start protocols are symmetrical. The central office (CO) and customer premise equipment (CPE) transmit identical bit patterns for each line state. Your application uses the same protocol file whether it functions as a CPE or emulates a CO.

The Loop Start and Ground Start protocols are asymmetrical. The CO and CPE transmit a different bit pattern to indicate the same condition. For example, the bit pattern the CO transmits to indicate idle is different from the bit pattern the CPE transmits to indicate idle. If you use these protocols, you load a different protocol file depending on whether your application acts as a CPE or CO.

If your system is in a loopback configuration using the Loop Start or Ground Start protocols, the trunk emulating the CO uses the CO protocol file and the trunk emulating the CPE uses the CPE protocol file.

In Tables 36 through 42, the values indicate the A and B bits transmitted by each party. The arrows point to the end receiving the bits. The driver has several parameters that control timing. For more information about the timing parameters, see the *RealCT Direct API Reference Manual*.

Immediate Start

Table 36 shows the pattern of bits transmitted by the CO and CPE in the Immediate Start protocol.

Table 36. Immediate Start: CO Calls CPE

Status	CO		CPE	Timing Parameter
Idle	00		00	
Seizure	11	-->	00	
CO sends digits	11		00	
Answer	11	<--	11	
Conversation	11		11	
CPE disconnects	11	<--	00	
CO disconnects	00	-->	00	*RDG_REMOTE_IDLE_TIMEOUT*
HookFlash	11	<--	11/00/11	*RDG_LOCAL_FLASH_DUR*
	11/00/11	-->	11	

In the Immediate Start protocol, both ends transmit 00 in the idle state. When the CO seizes the line, it transmits 11, and continues to transmit that pattern throughout the conversation. The CO waits a certain amount of time after seizing the line then sends digits, without confirming that the receiving end is ready. After the CO sends digits, the CPE answers the call by transmitting 11. To disconnect, both ends transmit 00.

The CO usually starts sending digits within 200 ms of the line seizure, so the application must be ready to receive digits almost immediately after receiving the seizure. When the application initiates a call, it sends digits 200 ms after seizing the line.

When the CPE disconnects, RHT_DISCONNECT or RHT_ON_HOOK waits for the CO to disconnect to a maximum time specified by *RDG_REMOTE_IDLE_TIMEOUT*.

If the network you are connected to accepts call transfers, the HookFlash duration is determined by *RDG_LOCAL_FLASH_DUR*.

Wink Start

Table 37 shows the pattern of bits transmitted by the CO and CPE in the Wink Start protocol.

Table 37. Wink Start: CO Calls CPE

Status	CO		CPE	Timing Parameters
Idle	00		00	
Seizure	11	-->	00	
Wink	11	<--	00/11/00	*RDG_REMOTE_MIN_SEIZE, RDG_LOCAL_WINK_DUR*
CO sends digits	11		00	
Answer	11	<--	11	
Conversation	11		11	
CPE disconnects	11	<--	00	
CO disconnects	00	-->	00	*RDG_REMOTE_IDLE_TIMEOUT*
HookFlash	11	<--	11/00/11	*RDG_LOCAL_FLASH_DUR*
	11/00/11	-->	11	

In the Wink Start protocol, both ends transmit 00 in the idle state. When the CO seizes the line, it transmits 11, and continues to transmit that pattern throughout the conversation. The CPE transmits a wink to acknowledge the seizure, then the CO begins sending digits. The CPE transmits 11 when it answers the call. To disconnect, both ends transmit 00.

The timing parameters for the wink depend on whether the application places the call or receives the call:

- When the application receives a call, it sends a wink after the time specified by *RDG_REMOTE_MIN_SEIZE*. The wink has a duration specified by *RDG_LOCAL_WINK_DUR*. The RHT_WAIT_LINE_ON function automatically sends the wink, so there is no need for the application to call RHT_SEND_WINK.
- When the application initiates a call, it waits for a wink for the time specified by *RDG_REMOTE_ACK_TIMEOUT*. The wink must have a duration between the minimum of *RDG_REMOTE_MIN_WINK* and the maximum *RDG_REMOTE_MAX_WINK*.

When the CPE disconnects, RHT_DISCONNECT or RHT_ON_HOOK waits for the CO to disconnect for a maximum time specified by *RDG_REMOTE_IDLE_TIMEOUT*.

If the network you are connected to accepts call transfers, the HookFlash duration is determined by *RDG_LOCAL_FLASH_DUR*.

When the application answers a call, it should not send the answer signal too soon after sending the wink. This should not be a problem if the application waits for digits, but a modified application that does not wait for digits after the wink should involve a delay time. The problem comes if there is too little time between the signals, so the application sends the pattern 00/11/00/11. The CO might ignore the second 00 because of its short duration and only acknowledge receiving the sequence 00/11. The CO interprets this as a very long wink and waits for it to end. Since the application has already sent the answer, it waits for digits from the CO.

When the application eventually hangs up, the CO considers the drop in the bits to be the end of the wink and waits for an answer. The application, on the other hand, has terminated the call and is ready to receive a new call. COs handle this situation differently. Some COs might consider this situation to be faulty signaling and block the circuit, while others continue waiting for the answer signal, producing a deadlock that prevents calls from being received in that circuit.

In the Wink Start protocol it is possible for the application to control when to send the wink. Since Wink Start is a derivative of Immediate Start, load the Immediate Start protocol then use RHT_SEND_WINK to send a wink in an inbound call or RHT_WAIT_WINK to receive a wink on an outbound call. Controlling the wink through the application can cause problems if the application gets preempted after RHT_SEIZE_LINE and before RHT_WAIT_WINK. The application does not receive any wink sent while the application was preempted since bits are only monitored while a function is running. With the Wink Start protocol loaded, the driver handles all timing issues. We recommend loading the appropriate Wink Start protocol rather than letting the application send or receive winks.

Double Wink Start

Table 38 shows the pattern of bits transmitted by the CO and CPE in the Double Wink Start protocol.

Table 38. Double Wink Start: CO Calls CPE

Status	CO		CPE	Timing Parameters
Idle	00		00	
Seizure	11	-->	00	
Wink	11	<--	00/11/00	*RDG_REMOTE_MIN_SEIZE, RDG_LOCAL_WINK_DUR*
CO sends digits	11		00	
Wink	11	<--	00/11/00	*RDG_REMOTE_MIN_SEIZE, RDG_LOCAL_WINK_DUR*
Answer	11	<--	11	
Conversation	11		11	
CPE disconnects	11	<--	00	
CO disconnects	00	-->	00	*RDG_REMOTE_IDLE_TIMEOUT*
HookFlash	11	<--	11/00/11	*RDG_LOCAL_FLASH_DUR*
	11/00/11	-->	11	

In the Double Wink Start Protocol, both ends transmit 00 in the idle state. When the CO seizes the line, it transmits 11 and continues to transmit that pattern throughout the conversation. The CPE transmits a wink to acknowledge the seizure, and another wink to acknowledge receiving digits. The CPE begins transmitting 11 when it answers the call. To disconnect, both ends transmit 00.

Double Wink Start is used mainly in Canada. It is a variation of the Wink Start Protocol in which the end receiving a call transmits a second wink after receiving digits. Since the timing for the second wink depends on the application, it requires a minor modification of the source code to explicitly send the second wink after receiving the digits.

The timing parameters for the wink depend on whether the application places the call or receives the call:

- When the application receives a call, it sends a wink after the time specified by *RDG_REMOTE_MIN_SEIZE*. The wink has the duration specified by *RDG_LOCAL_WINK_DUR*. The second wink follows the same timing.
- When the application initiates a call, it waits for a wink within the time specified by *RDG_REMOTE_ACK_TIMEOUT*. The wink must have a duration between the minimum of *RDG_REMOTE_MIN_WINK* and the maximum *RDG_REMOTE_MAX_WINK*. After dialing out, the application calls RHT_WAIT_WINK to detect the second wink. The duration of the second wink must be in the same range as the first, but RHT_WAIT_WINK does not have a maximum time to run. The application terminates the function after a reasonable time has elapsed.

When the CPE hangs up, RHT_DISCONNECT or RHT_ON_HOOK waits for the CO to hang up for a maximum time specified by *RDG_REMOTE_IDLE_TIMEOUT*.

If the network you are connected to accepts call transfers, the HookFlash duration is determined by *RDG_LOCAL_FLASH_DUR*.

As in the Wink Start protocol, be sure to include a delay between the second wink and the answer so the remote end receives the wink properly.

Loop Start

Tables 39 and 40 show the pattern of bits transmitted by the CO and the station in the Loop Start protocol. Table 39 shows the pattern when the CO calls the station.

Table 39. Loop Start: CO Calls Station

Status	CO		Station	Timing Parameter
Idle	01		01	
Seizure: Ringing	00	-->	01	
Answer	01	<--	11	
Conversation	01		11	
Station disconnects	01	<--	01	
HookFlash	01	<--	11/01/11	*RDG_LOCAL_FLASH_DUR*

In the Loop Start Protocol, both ends transmit 01 in the idle state. When the CO calls the station, it transmits 00 then returns to 01 when the station sends the 11 answer signal. The station disconnects by returning to the 01 state.

The CO indicates a ring pattern by toggling the B bit from 0 to 1. The cadence is the same as the ring for analog lines, 2000 ms ring present, 4000 ms ring absent, where 0 indicates ring present and 1 indicates ring absent.

The CO does not send digits to the station unless the line is configured to provide Caller ID information. There is no information when the calling party disconnects.

Table 40 shows the pattern of bits transmitted when the station calls the CO.

Table 40. Loop Start: station Calls CO

Status	CO		Station	Timing Parameter
Idle	01		01	
Seizure	01	<--	11	
Dialtone from CO	01		11	
Digits from Station	01		11	
Answer	01		11	
Conversation	01		11	
Station disconnects	01	<--	01	
HookFlash	01	<--	11/01/11	*RDG_LOCAL_FLASH_DUR*

In the Loop Start protocol, both ends transmit 01 in the idle state. When the station calls the CO, the CO continuously sends a 01 signal. The station seizes the line using a 11 signal, then disconnects by returning to the idle 01 state.

When the station disconnects, RHT_DISCONNECT or RHT_ON_HOOK terminates immediately, since there is no bit change that the CO needs to perform.

When the application initiates a call, it waits for a dial tone, then dials digits. There is no signal from the CO to acknowledge the outgoing call, indicate an answer by the called party, or indicate that the called party disconnected.

If the network you are connected to accepts call transfers, the HookFlash duration is determined by *RDG_LOCAL_FLASH_DUR*.

Ground Start

Tables 41 and 42 show the pattern of bits transmitted by the CO and station in the Ground Start protocol. Table 39 shows the pattern when the CO calls the station.

Table 41. Ground Start: CO Calls Station

Status	CO		Station	Timing Parameter
Idle	11		01	
Seizure: Ground on Tip	01	-->	01	
Ringing	00	-->	01	
Answer	01	<--	11	
Conversation	01		11	
Station disconnects	01	<--	01	
CO disconnects	11	-->	01	*RDG_REMOTE_IDLE_TIMEOUT*
HookFlash	01	<--	11/01/11	*RDG_LOCAL_FLASH_DUR*
	01/11/01	-->	11	

In the Ground start protocol, the CO indicates idle with a 11 pattern, and the station indicates idle with a 01. The CO seizes the line with a 01 pattern, then rings by toggling the B bit from 0 to 1. The cadence is the same as the ring for analog lines, 2000 ms ring present, 4000 ms ring absent, where 0 indicates ring present and 1 indicates ring absent. When the station answers with a 11 pattern, the CO transmits a 01 for the remainder of the call. The station and CO disconnect by returning to the idle state.

The CO does not send digits to the station unless the line is configured to provide Caller ID information. When the station hangs up, RHT_DISCONNECT or RHT_ON_HOOK waits for the CO to hang up for a maximum time specified by *RDG_REMOTE_IDLE_TIMEOUT*.

Table 42 shows the pattern of bits transmitted when the station calls the CO.

Table 42. Ground Start: Station Calls CO

Status	CO		Station	Timing Parameter
Idle	11		01	
Seizure: Ground on Ring	11	<--	00	
Ground on tip from CO	01	-->	00	
Station removes ground	01	<--	11	
Dialtone from CO	01		11	
Digits from Station	01		11	
Answer	01		11	
Conversation	01		11	
CO disconnects	11	-->	01	
HookFlash	01	<--	11/01/11	*RDG_LOCAL_FLASH_DUR*
	01/11/01	-->	11	

In the Ground start protocol, the CO indicates idle with a 11 pattern, and the station indicates idle with a 01. The station seizes the line by transmitting a 00, and the CO acknowledges by transmitting 01, which it continues to transmit throughout the call. The station then transmits 11, which it transmits throughout the call. This handshaking in built into RHT_SEIZE_LINE and RHT_OFF_HOOK so there is no need for the application to be involved in the handshaking process. The CO then provides dial tone to indicate that it is ready to receive digits. There is no signal from the CO to indicate an answer by the called party. The CO or station disconnects by returning to the idle state.

If the network you are connected to accepts call transfers, the HookFlash duration is determined by *RDG_LOCAL_FLASH_DUR*.

Appendix B
E1 Line Protocols

This appendix describes the signaling bits transmitted in the most common E1 line protocols.

It includes the following sections:

- Overview of Protocols
- R2-CCITT
- R2-CCITT - Chinese Implementation
- R2-CCITT - Brazilian Implementation
- R2-CCITT - Central European Implementation

Overview of Protocols

The R2-CCITT protocols are symmetrical. The central office (CO) and customer premise equipment (CPE) transmit identical bit patterns for each line state. Your application uses the same protocol whether it functions as a CPE or emulates a CO.

In Tables 43 through 44, the values indicate the A and B signaling bits transmitted by each party. The arrows point to the end receiving the bits. The driver has several parameters that control timing. For more information about the timing parameters, see the *RealCT Direct API Reference Manual*.

R2-CCITT

Table 43 shows the pattern of bits transmitted by the CO and CPE in the R2-CCITT protocol.

Table 43. R2-CCITT: CO Calls CPE

Status	CO		CPE	Timing Parameter
Idle	10		10	
Seizure	00	-->	10	
Acknowledgment	00	<--	11	*RDG_LOCAL_MIN_SEIZURE_ACK, RDG_LOCAL_ACK_GUARD_TIME*
R2 Inter-register Signaling	00		11	
Answer	00	<--	01	*RDG_LOCAL_ANSWER_DUR, RDG_LOCAL_ANSWER_GUARD_TIME*
Conversation	00		01	
Clear Back	00	<--	11	*RDG_LOCAL_CLEAR_BACK_DUR*
Clear Forward	10	-->	11	*RDG_LOCAL_CLEAR_BACK_DUR*
Release	10	<--	10	
Blocked	10		11	
Hook Flash	00	<--	01/11/01	*RDG_LOCAL_FLASH_DUR*
	00/10/00	-->	01	
Forced Release	00	<--	00	

In the R2-CCITT line protocol, bits C and D are fixed, C=0, D=1. Both ends transmit 10 in the idle state. The CO seizes the line with 00, which it continues to transmit throughout the call. The CPE acknowledges the seizure with 11. It continues to transmit 11 through the inter-register signaling, then transmits 01 to answer the call. At the end of the call, the CPE transmits a 11 to clear the line, then returns to idle.

All bit transitions are recognized after the time specified by RDG*_LINE_DEGLITCH_TIME* parameter.

When the application is the incoming end, the application immediately recognizes the incoming call and the protocol sends an acknowledgment signal for at least *RDG_LOCAL_MIN_SEIZURE_ACK*. Then, the function waits for *RDG_LOCAL_ACK_GUARD_TIME* and returns.

To answer a call, the protocol sends the answer signal for at least *RDG_LOCAL_ANSWER_DUR*, then the function waits for *RDG_LOCAL_ANSWER_GUARD_TIME* and returns.

The application can disconnect a call using a Clear Back or a Forced Release. The Forced Release signal causes fewer delays on the line, but is not supported by all carriers. To disconnect using a Clear Back, the signal must be present on the line for at least *RDG_LOCAL_CLEAR_BACK_DUR*. When the application receives the Clear Forward (or after *RDG_REMOTE_IDLE_TIMEOUT*, whichever happens first), the function waits for *RDG_LOCAL_CLEAR_BACK_GUARD_TIME* and returns.

To disconnect using a Forced Release, the signal must be present on the line for at least *RDG_LOCAL_CLEAR_BACK_DUR*. When the application detects the Clear Forward signal (or after *RDG_REMOTE_IDLE_TIMEOUT*, whichever happens first), the function waits for *RDG_LOCAL_CLEAR_BACK_GUARD_TIME* and returns.

If the application is the outgoing end, it seizes the line, waits for an acknowledgment for a time defined by *RDG_REMOTE_ACK_TIME*, then waits *RDG_LOCAL_SEIZE_GUARD_TIME* and returns. If the application does not receive the seizure acknowledgment, it transmits an idle signal and the function returns.

A disconnect signal (Clear Back, Forced Release or Clear Forward, depending on whether the application is the incoming or outgoing end), must be present on the line for at least *RDG_REMOTE_MIN_DISCONNECT*. A Clear Back present on the line after a clear forward is sent for more than *RDG_REMOTE_BLOCK_MIN* is considered to be a Block signal.

The Hook Flash signal has the duration specified by *RDG_LOCAL_FLASH_DUR*. The guard time at the end of the function is specified by *RDG_LOCAL_FLASH_GUARD_TIME*.

R2-CCITT - Chinese Implementation

The Chinese implementation of the R2-CCITT protocol uses the same timing and bit patterns as the R2-CCITT shown in Table 43. The only difference is that signaling bits C and D are both set to 1.

R2-CCITT - Brazilian Implementation

The Brazilian implementation of the R2-CCITT protocol uses the same timing and bit patterns as the R2-CCITT, except the receiving end answers the line by transmitting 01/11/01. This pattern is an answer, followed by a Clear Back, then a second answer. The receiving end transmits the answer for *RDG_LOCAL_DOUBLE_ANSWER_ON*. It transmits the Clear Back for *RDG_LOCAL_DOUBLE_ANSWER_OFF*. The Clear Back signal in this double answer procedure disconnects any collect calls.

R2-CCITT - Central European Implementation

The Central European implementation of the R2-CCITT protocol uses the same timing and bit patterns as the R2-CCITT but with a different disconnect procedure, as shown in Table 44.

Table 44. R2-CCITT Central European: CO Calls CPE

Status	CO		CPE
Idle	10		10
Seizure	00	-->	10
Acknowledgment	00	<--	11
R2 Inter-register Signaling	00		11
Answer	00	<--	01/11/01
Conversation	00		01
Clear Back	00	<--	11
Clear Forward *	10	-->	11
Clear Forward	10	-->	01
Release Guard	10	<--	11
Release **	10	<--	10
Blocked	10		11
Hook Flash	00	<--	01/11/01
	00/10/00	-->	01

(*) Incoming end initiating disconnect.

(**) Outgoing end initiating disconnect.

When the outgoing end terminates the call, it sends a Clear Forward signal. When the incoming end detects the Clear Forward, it transmits an intermediate signal called a Release Guard before releasing the line. The Release Guard acts as an intermediate signal to prevent bits A and B from both changing at one time. An Answer Signal (01) followed by an Idle signal (10) requires both bits A and B to change, called a double bit change. By using a Release Guard (11), only one bit changes at a time. Most implementations of R2-CCITT interpret a Release Guard as a temporary Block signal on the line. Although this transition is not necessary in other implementations, it does not cause problems.

Index

Symbols

A

B

C

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

S

T

U

U

V

W

Y

Z