OmniBus CSDB
USER'S MANUAL

for BTIDriver-Compliant Devices

July 15, 2005
Rev. A

Ballard Technology
3229A Pine Street
Everett, WA 98201 USA

Phone: (800) 829-1553 (425) 339-0281 Fax: (425) 339-0915
E-mail: *support@ballardtech.com*
Web: *www.ballardtech.com*

MA146-071505

COPYRIGHT NOTICE

SAFETY WARNING

Ballard products are of commercial grade and therefore are neither designed, manufactured, nor tested to standards required for use in critical applications where a failure or deficiency of the product may lead to injury, death, or damage to property. Without prior specific approval in writing by the president of Ballard Technology, Inc., Ballard products are not authorized for use in such critical applications.

TRADEMARKS

Molex® LFH™ is a trademark of Molex Inc. OmniBus® is a registered trademark of Ballard Technology, Inc. BTIDriver™ and OmniBusBox™ are trademarks of Ballard Technology, Inc. All other product names or trademarks are property of their respective owners.

Ballard Technology 3229A Pine Street, Everett, WA 98201 USA
Phone: (800) 829-1553 (425) 339-0281 Fax: (425) 339-0915
E-mail: *support@ballardtech.com* Web: *www.ballardtech.com*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

This page intentionally blank.

1. INTRODUCTION

OmniBus® CSDB interfaces provide Commercial Standard Digital Bus serial communications for Ballard's family of OmniBus avionics databus interface products. OmniBus CSDB may be used for communicating with and monitoring serial digital avionics equipment including communication and navigation radios. The CSDB products also provide general-purpose scheduled and asynchronous serial communications using RS-422, RS-232, and RS-423 serial interface standards.

This manual describes the hardware connection and software programming of OmniBus CSDB interfaces.

1.1 CSDB Overview

The Commercial Standard Digital Bus (CSDB) is the specification that defines the transmission of serial digital data for avionics equipment. The specification describes how an avionics system transmits information over a single twisted and shielded pair of wires (the databus) using devices complying with EIA RS-422A. The CSDB standard can be purchased through the General Aviation Manufacturers Group (GAMA) at www.gama.aero.

1.2 Ballard OmniBus CSDB Devices

The table below lists the OmniBus I/O modules available with CSDB capability.

Part No.	Description
433	4R/4T CSDB
437	4R/4T CSDB with 4R/4T ARINC 429 channels

Both the 433 and 437 modules have four receive and four transmit CSDB channels. All CSDB receive channels feature independent address byte and source identifier (SI) filtering. Each transmit channel automatically maintains accurate message repetition rates and can transmit continuous, non-continuous, and burst message types. All CSDB channels may operate at standard low or high speed (12.5 or 50 Kbps) bit rates as well as programmable bit rates. CSDB channels may also be used for general-purpose serial communication using the RS-422, RS-232, and RS-423 interface standards.

Module 437 includes four receive and four transmit ARINC 429 channels. All ARINC 429 receive channels feature automatic speed detection and independent label and SDI filtering. Each transmit channel automatically maintains accurate label repetition rates. To support data transfer protocols, aperiodic words may be transmitted without altering the timing of periodic words. Both receive and transmit channels may be independently set for standard low or high speed (12.5 or 100 Kbps). Operation of the ARINC 429 channels is documented in other manuals.

Ballard's OmniBus CSDB interfaces are available for the entire OmniBus family of PCI, cPCI, VME, and OmniBusBox™ platforms.

1.3 Software Functions for CSDB Communications

BTIDriver™ is a unified library of API (Application Program Interface) functions designed to control Ballard's BTIDriver-compliant hardware products. Ballard Technology makes many hardware products that interface with various avionics databuses or other interfaces to facilitate development, simulation, and testing. Software is used to operate these hardware devices. Programmers can use BTIDriver to create custom software for Ballard hardware devices.

BTIDriver supports many interface standards and a variety of Ballard hardware devices. As long as the devices have similar hardware capability, BTIDriver applications written for one device can run on another device with little or no change. This manual only documents the principles and functions used for CSDB communications software applications (see Chapter 3, Chapter 4 and Appendix A). Other protocols are documented separately. Appendix B (Multi-protocol/Device Programs) provides guidance for writing multi-protocol BTIDriver applications.

1.4 How to Use This Manual

The OmniBus CSDB User's Manual is designed to be both a tutorial and a reference guide. Chapter 2 (Hardware Interface) tells you how to make the physical connection to CSDB and other serial buses. After reading Chapter 3 (Programming Basics) and referring to Appendix A (CSDB Function Reference), you should be able to write simple computer programs to operate your Ballard CSDB interface device. Refer to Chapter 4 (Advanced Operation) for more complex applications.

Your OmniBus platform (PCI/cPCI, VME, or OmniBusBox) user's manual and other protocol (MIL-STD-1553, ARINC 429, etc.) programming manuals can provide you with additional useful information.

This manual assumes that you are familiar with the essentials of compiling, linking, and running programs in C. With minor exceptions, the content of this manual also applies to other programming languages. It is also assumed that you are familiar with the CSDB standard.

The following conventions are observed throughout this manual:

1. "Device" with a capital "D" is used generically to mean any Ballard CSDB BTIDriver-compliant interface device. This includes OmniBus PCI, cPCI, VME, and OmniBusBox platforms.
2. Driver function names are in **bold** type and are all prefixed by "**`BTICard_`**" or "**`BTICSDB_`**" (e.g., **`BTICSDB_ChConfig`**). **`BTICSDB_`** is the prefix for all CSDB function names regardless of whether they are used in CSDB or general-purpose serial modes.
3. A small "h" suffix indicates hexadecimal values (e.g., F01Ch).
4. Constants defined in the driver software are written in all capital letters (e.g., CHCFGCSDB_DEFAULT).
5. The symbol "??" is used in function names in this manual to indicate a category of functions with similar uses or attributes. These characters should be replaced by a category prefix or suffix in an actual function call (e.g.,

BTICSDB_MsgData?? to represent **BTICSDB_MsgDataRd** and **BTICSDB_MsgDataWr**).

1.5 Technical Support and Customer Service

Ballard Technology offers technical support before and after purchase. Our hours are 9:00 am to 5:00 pm Pacific Time, though support and sales engineers are often available outside those hours. We invite your questions and comments on any of our products. You may reach us by telephone at (800) 829-1553 or (425) 339-0281, by fax at (425) 339-0915, on the Web at www.ballardtech.com, or through e-mail at support@ballardtech.com.

1.6 Updates

At Ballard Technology, we take pride in high-quality, reliable products that meet the needs of our customers. Because we are continually improving our products, periodic updates to documentation and software may be issued. Please visit us at www.ballardtech.com and register your product(s) so that we can keep you informed of updates, customer services, and new product information.

This page intentionally blank

2. HARDWARE INTERFACE

Part of the CSDB standard specifies an electrical interface. CSDB communication is asynchronously broadcast over an RS-422 electrical interface. The OmniBus CSDB product is capable of transmitting and receiving over RS-422, RS-232, and RS-423 electrical interfaces for both CSDB and general-purpose serial communication.

The basic differences between CSDB and other general-purpose serial communication modes are the transmission lines and voltage levels. You can set up and operate each OmniBus CSDB channel with a different electrical interface by the way you wire it and configure it in software.

This chapter shows you how to wire the electrical interface for CSDB and the various general-purpose serial modes, and it explains the difference. Chapter 3 and Appendix A (see the **`BTICSDB_ChConfig`** function) show you how to configure the electrical interface of the channels in software.

2.1 CSDB Standard

The CSDB standard specifies the use of an RS-422 electrical interface. RS-422 is a serial communications standard that uses balanced differential signals. RS-422 can be implemented in either a point-to-point (one driver and one receiver, as shown in Figure 2.1) or a multidrop (one driver and up to 10 receivers) architecture. The transmission line should be a twisted shielded pair for best results.

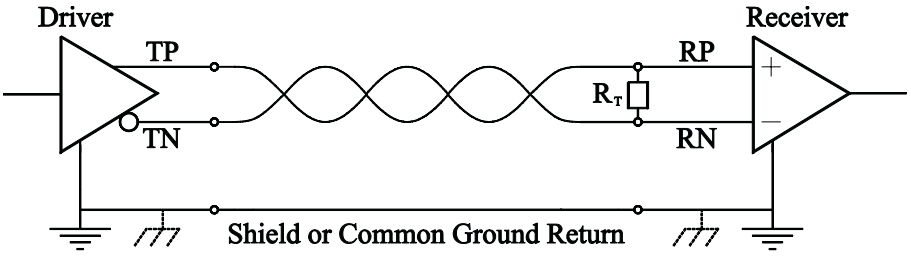


Figure 2.1—CSDB and RS-422 bus wiring

OmniBus CSDB interfaces are implemented with differential drivers and receivers that should be wired for RS-422 as shown in Figure 2.1 – driver positive (TP) to receiver positive (RP), driver negative (TN) to receiver negative (RN), and grounds (often a shield) connected between the driver and receiver. The ground between the driver and receiver(s) is required. CSDB does not require the use of a terminating resistor (R_T) due to the low speed of the bus.

2.2 General-Purpose Serial Standards

The OmniBus CSDB product provides general-purpose serial communication on any of its channels using RS-422, RS-232, and RS-423. This section shows how your CSDB device should be wired to accommodate the various serial communication modes.

2.2.1 RS-422 (Balanced line)

General-purpose serial communication for RS-422 should be wired as described for CSDB in Section 2.1. A terminating resistor is recommended at the end of each twisted pair transmission line when using high speed data rates (above 200 Kbps) or long cables. When used, the terminating resistor must be added external to the OmniBus CSDB interface. The terminating resistor should match the characteristic impedance of the cable.

2.2.2 RS-232 and RS-423 (Unbalanced line)

RS-232 and RS-423 are serial communications standards that use unbalanced (single line with a ground reference) signals. RS-232 buses are wired as shown in Figure 2.2a. These standards only operate in a point-to-point architecture.

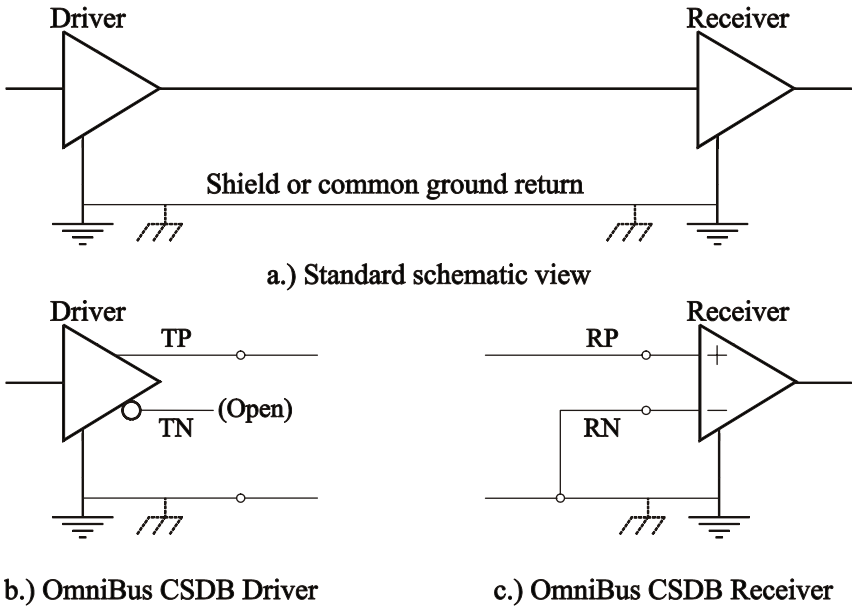


Figure 2.2—RS-232 and RS-423 bus wiring

The balanced line drivers and receivers on OmniBus CSDB devices can be configured to work with unbalanced lines. To use an OmniBus CSDB driver with an RS-232 or RS-423 unbalanced line connect the driver positive output (TP) to the receiver's input signal and leave the negative driver output (TN) open. Also, connect a ground between the driver and receiver (See Figure 2.2b)

To use the OmniBus CSDB device as an unbalanced line receiver connect the driver output to the OmniBus receiver's positive input (RP) and ground the negative input (RN). In all cases the signal ground must be connected between drivers and receivers (See Figure 2.2c). Multiple unbalanced lines often share a common ground.

2.3 Connector Pin Numbers

This section provides the correlation between channels and connector pin numbers on OmniBus CSDB devices and standard cables. Use the principles described in Section 2.1 and Section 2.2 to wire each channel for balanced or

unbalanced transmission lines. Remember all serial standards require a ground to be connected between the transmitting system and the receiving system.

The standard connector on OmniBus products is a 60-pin Molex® LFH™ receptacle, which have the basic pin designations shown in Table 2.1. Subsections below describe how these pin designations apply to OmniBus CSDB modules and associated cables. Note that wiring is done in pairs; 30 pairs total. On balanced signals (especially databuses, labeled "BUSxx" in Table 2.1), be sure to use twisted pairs to avoid cross talk. The suffix on the designations for databus signals in Table 2.1 represents the polarity (P for positive and N for negative). Refer to your OmniBus platform (PCI/cPCI, VME, or OmniBusBox) user's manual for additional information on the LFH connector, discretes, and IRIG.

Pair #	LFH Pin	Name	Pair #	LFH Pin	Name
1	2	BUS0N	16	32	BUS8N
1	3	BUS0P	16	33	BUS8P
2	4	BUS2N	17	34	BUS10N
2	5	BUS2P	17	35	BUS10P
3	6	BUS4N	18	36	BUS12N
3	7	BUS4P	18	37	BUS12P
4	8	BUS6N	19	38	BIS14N
4	9	BUS6P	19	39	BUS14P
5	10	GND	20	40	GND
5	11	CDIN0A	20	41	BDIN0
6	12	GND	21	42	GND
6	13	CDOUT0A	21	43	BDOUT0
7	14	CGND	22	44	CGND
7	15	NC5VA	22	45	NC5VA
8	16	GND	23	46	GND
8	17	IRIG	23	47	IRIG
9	18	GND	24	48	GND
9	19	CDOUT1A	24	49	CDOUT2A
10	20	GND	25	50	GND
10	21	CDIN1A	25	51	CDIN2A
11	22	BUS7P	26	52	BUS15P
11	23	BUS7N	26	53	BUS15N
12	24	BUS5P	27	54	BUS13P
12	25	BUS5N	27	55	BUS13N
13	26	BUS3P	28	56	BUS11P
13	27	BUS3N	28	57	BUS11N
14	28	BUS1P	29	58	BUS9P
14	29	BUS1N	29	59	BUS9N
15	30	GND	30	60	GND
15	1	GND	30	31	GND

Table 2.1—General pin designations

2.3.1 CSDB Modules

Each CSDB module has four receive and four transmit channels. Some CSDB modules also have ARINC 429 channels.

Table 2.2 shows the pin assignments for the LFH connector and the 16036 accessory cable for different OmniBus CSDB modules. The CSDB standard, all the serial standards, and ARINC 429 require a ground connection between the transmitting system and the receiving system.

433 Module	437 Module	Transmit/ Receive	Channel/ Port	Signal Name	Polarity	LFH Pin#	16036 D-sub Pin#	LFH Name
n/a	ARINC 429	R	CH0	CH0P	+	3	P2-3	BUS0P
				CH0N	-	2	P2-15	BUS0N
	ARINC 429	R	CH1	CH1P	+	28	P2-2	BUS1P
				CH1N	-	29	P2-14	BUS1N
	ARINC 429	R	CH2	CH2P	+	5	P2-4	BUS2P
				CH2N	-	4	P2-16	BUS2N
	ARINC 429	R	CH3	CH3P	+	26	P2-5	BUS3P
				CH3N	-	27	P2-17	BUS3N
CSDB	CSDB	R	CH4	CH4P	+	7	P2-6	BUS4P
				CH4N	-	6	P2-18	BUS4N
CSDB	CSDB	R	CH5	CH5P	+	24	P2-8	BUS5P
				CH5N	-	25	P2-19	BUS5N
CSDB	CSDB	R	CH6	CH6P	+	9	P2-9	BUS6P
				CH6N	-	8	P2-20	BUS6N
CSDB	CSDB	R	CH7	CH7P	+	22	P2-10	BUS7P
				CH7N	-	23	P2-21	BUS7N
n/a	ARINC 429	T	CH8	CH8P	+	33	P3-3	BUS8P
				CH8N	-	32	P3-15	BUS8N
	ARINC 429	T	CH9	CH9P	+	58	P3-2	BUS9P
				CH9N	-	59	P3-14	BUS9N
	ARINC 429	T	CH10	CH10P	+	35	P3-4	BUS10P
				CH10N	-	34	P3-16	BUS10N
	ARINC 429	T	CH11	CH11P	+	56	P3-5	BUS11P
				CH11N	-	57	P3-17	BUS11N
CSDB	CSDB	T	CH12	CH12P	+	37	P3-6	BUS12P
				CH12N	-	36	P3-18	BUS12N
CSDB	CSDB	T	CH13	CH13P	+	54	P3-8	BUS13P
				CH13N	-	55	P3-19	BUS13N
CSDB	CSDB	T	CH14	CH14P	+	39	P3-9	BUS14P
				CH14N	-	38	P3-20	BUS14N
CSDB	CSDB	T	CH15	CH15P	+	52	P3-10	BUS15P
				CH15N	-	53	P3-21	BUS15N

*Signal grounds are available on the following LFH pins: 1, 10, 12, 16, 18, 20, 30,31, 40, 42, 46, 48, 50, 60

*Signal grounds are available on the following 16036 D-sub pins: P2-1, P2-23, P3-1, P3-23

Table 2.2—Pinout of CSDB modules to LFH and 16036 D-sub connectors

2.3.2 Standard Cables

Ballard sells a number of different cables that are useful for wiring to OmniBus products. Each cable has a standard length. Non-standard lengths may be specified by adding a /xx suffix after the part number, where xx is the length in feet. For example, a 16036/10 is a ten foot long 16036.

2.3.2.1 PN 16035 cable assembly: LFH to LFH

This is a three-foot-long straight-through cable with 60-pin male LFH plugs on both ends. It is wired pin for pin and pair for pair as shown in Table 2.1. The 16035 is useful for connecting an OmniBus product to a user-provided panel or other assembly.

2.3.2.2 PN 16036 cable assembly: LFH to two 25-pin D-subs

This is a three-foot-long Y-cable that adapts a 60-pin male LFH plug (labeled P1) to two 25-pin D-sub connectors (P2 and P3). Because of the size and popularity of D-sub connectors, some users may find it easier to interface to them than to the OmniBus LFH connectors. As can be seen from Table 2.3, there is symmetry between the upper and lower halves of the LFH connector. On the 16036 cable assembly, the upper half of the LFH connector is wired to one D-sub and the lower half is wired to the other D-sub, thus giving similar signals on the corresponding pins of both D-subs. The wire pairs on the 16036 are different from those on the 16035. Wiring for the 16036 cable with OmniBus CSDB module signal names is shown in Table 2.3 below.

P2 Pair #	From P1 pin	To P2 pin	Signal Name	LFH Name
1	3	3	CH0P	BUS0P
1	2	15	CH0N	BUS0N
2	28	2	CH1P	BUS1P
2	29	14	CH1N	BUS1N
3	5	4	CH2P	BUS2P
3	4	16	CH2N	BUS2N
4	26	5	CH3P	BUS3P
4	27	17	CH3N	BUS3N
5	7	6	CH4P	BUS4P
5	6	18	CH4N	BUS4N
6	24	8	CH5P	BUS5P
6	25	19	CH5N	BUS5N
7	9	9	CH6P	BUS6P
7	8	20	CH6N	BUS6N
8	22	10	CH7P	BUS7P
8	23	21	CH7N	BUS7N

P3 Pair #	From P1 pin	To P3 pin	Signal Name	LFH Name
1	33	3	CH8P	BUS8P
1	32	15	CH8N	BUS8N
2	58	2	CH9P	BUS9P
2	59	14	CH9N	BUS9N
3	35	4	CH10P	BUS10P
3	34	16	CH10N	BUS10N
4	56	5	CH11P	BUS11P
4	57	17	CH11N	BUS11N
5	37	6	CH12P	BUS12P
5	36	18	CH12N	BUS12N
6	54	8	CH13P	BUS13P
6	55	19	CH13N	BUS13N
7	39	9	CH14P	BUS14P
7	38	20	CH14N	BUS14N
8	52	10	CH15P	BUS15P
8	53	21	CH15N	BUS15N

Table 2.3—Wiring chart for 16036 cable assembly (continued on next page)

P2 Pair #	From P1 pin	To P2 pin	Signal Name	LFH Name
9	11	11	CDIN0	
9	10	23	GND	
10	17	12	IRIG	
10	19	24	CDOUT1	
11	13	13	CDOUT0	
11	15	25	NC5V	
12	21	22	CDIN1	
12	20	1	GND	
13	14	7	CGND	

P3 Pair #	From P1 pin	To P3 pin	Signal Name	LFH Name
9	51	11	BDIN0/1*	
9	40	23	GND	
10	47	12	IRIG	
10	49	24	CDOUT2	
11	43	13	BDOUT0/1*	
11	45	25	NC5V	
12	51	22	CDIN2	
12	50	1	GND	
13	44	7	CGND	

*0 or 1, depending on core A or B

Braids connected shell to shell

See your OmniBus platform user's manual for information about discrete and IRIG signals.

Table 2.3—Wiring chart for 16036 cable assembly (continued)

3. PROGRAMMING BASICS

This chapter illustrates basic CSDB operation using the BTIDriver library. Examples demonstrate:

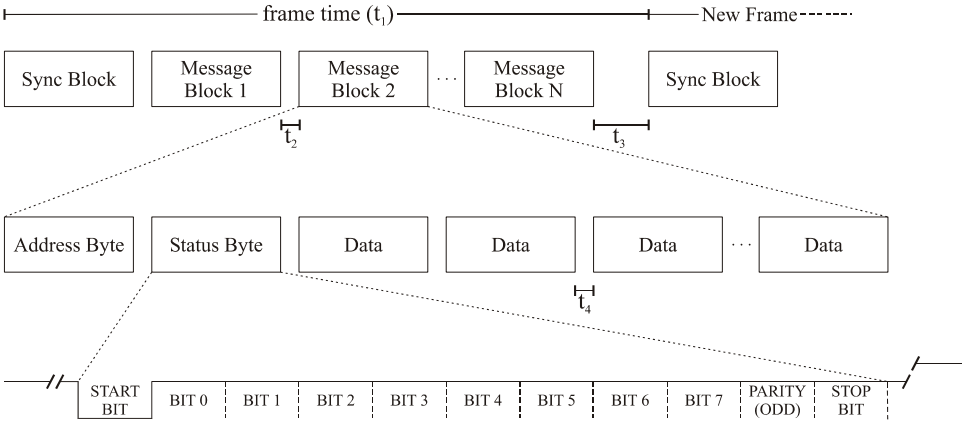
- operation of CSDB transmit channels
- operation of CSDB receive channels
- monitoring and selectively recording CSDB bus traffic

The examples provide code fragments in the C language. Libraries for other standard languages are available on request from Ballard Technology. Driver functions are prefixed by "**`BTICard`**_" for protocol-independent Device functions and "**`BTICSDB`**_" for CSDB-specific functions. The examples use CH4 as the receive channel and CH12 as the transmit channel. The transmit and receive channels for your CSDB Device may be different.

This chapter describes the operational elements relevant to most applications. After reading this chapter, you will be familiar with the essentials of an application program, and you will recognize the most important driver functions. Complete descriptions of all driver functions may be found in Appendix A.

3.1 Terminology

A CSDB bus consists of repeated frames of data divided into message blocks. Each message block contains a particular number of bytes including an address byte, a status byte, and additional data bytes. Messages can be of type continuous, non-continuous, or burst. A CSDB bus definition specifies the number of frames per second, message blocks per frame, bytes per message block, and the bit rate. Figure 3.1 shows a typical bus structure including frames, message blocks, and byte oriented data.



t_1 = frame time (1/max update)
t_2 = interblock time (no restrictions)
t_3 = bus idle time (11 bit-times minimum)
t_4 = interbyte time (no restrictions)

Figure 3.1—Typical Bus Structure

3.1.1 Frame

A frame is defined as the interval from the start of a sync block to the start of the next sync block. The number of message blocks per frame is specified in the characteristics of the particular CSDB bus. Every frame must begin with a sync message even if no other messages are present in that frame.

3.1.2 Message Block

A message block is defined as a single, serial message consisting of a fixed number of bytes transmitted in a fixed sequence. The number of bytes per message block is specified in the characteristics of the particular CSDB bus. A message block may also be referred to as a message. Each message is identified by the first byte in the sequence which is called the address byte (or label). The second byte contains status and/or state information including source identifier (SI) bits. The second byte may also be used as data instead of status. Any remaining bytes contain data or are padded to logic "0". A sync message is a special message with all bytes set to A5h.

Each message may be configured as a continuous, non-continuous, or burst message.

Continuous: Each message is repeated at the specified update rate.

Non-continuous: Each message is repeated at the specified update rate when it is available, but may not be available at all times.

Burst: Each message is repeated in each of sixteen consecutive frames when it is available.

See the CSDB specification for specific message block definitions.

3.1.3 Byte

CSDB data is transmitted in byte format over an inverted RS-422 electrical interface. Each byte consists of a start bit (logic "0"), 8 data bits, an odd parity bit, and a stop bit (logic "1").

3.1.4 Gap

The default idle state of the bus is logic level "1". Idle gap time between bytes and message blocks is undefined as long as the message update rates and bus parameters (i.e., frames per second, blocks per frame, bytes per block) are maintained. A minimum of 11 bit-times of gap is required between frames.

3.2 Getting Started

The first step in developing an application is identifying the CSDB bus parameters and messages to be handled. The following examples are to illustrate programming concepts and are not intended to represent actual CSDB bus or message definitions. For the examples in this chapter, suppose we want to

- simulate transmission of a continuous message and a burst message.
- receive continuous and burst messages.
- monitor all bus activity and record it to disk for later analysis.

Specifically, assume we want to interface with a bus defined by the parameters shown in Table 3.1, and we are interested in the messages defined in Table 3.2. In an actual application the bus parameters and message definitions would be found in an equipment manual or the CSDB specification.

Bus Parameter	Value
Bit Rate	12.5 kbps
Bytes per Block	6
Frames per Second	10
Blocks per Frame (max)	18

Table 3.1—Example CSDB Bus Parameters

Message Description	Address (HEX)	Updates/Second	Message Type
Sync	A5	10	Continuous
Current Frequency	12	5	Continuous
Remote Tune	F3	0 or 10	Burst

Table 3.2—Example CSDB Messages

The Sync message is a continuous message type that is transmitted in every frame and is used to allow receivers to synchronize to the CSDB bus. The Current Frequency message is a continuous message type that is transmitted in every other frame. For these examples, the data in the Current Frequency message is 121.1MHz. The Remote Tune message is a burst message type that is transmitted in sixteen consecutive frames every time the message is set valid for transmission. For these examples, the data in the Remote Tune message is used to remotely tune a radio to 118.9MHz. Note that the data in the examples are implied to be in the 100MHz range; therefore, only the lower digits are transmitted.

Part of the configuration process is associating messages with Message Records. Each Message Record contains an array of data, including the address byte, the status byte, and any additional data bytes. The Message Record may also include some extra data related to that message (e.g., the time-tag). When the Device receives a given message, the message is stored in the data array of the designated Message Record. When the Device transmits a given message, the data to be transmitted is retrieved from the data array of the Message Record. More detailed information on Message Records may be found in Section 4.2.

3.3 Steps a Program Must Perform

Before examining the examples, you should understand the sequence of steps a program must perform to operate a Device. Operating your Device with the BTIDriver library involves seven general steps, four of which require only a single function call. Three of the steps involve many options, so the number of function calls required depends on which options are desired. The steps are as follows:

1. **Open: `BTICard_CardOpen`** is always the first function called, followed by a call to **`BTICard_CoreOpen`** for each core (see Appendix B for a complete discussion of card and core handles). It is good programming

practice to use **`BTICard_CardReset`** (for each core) before calling configuration functions. This clears the core and eliminates all residual configuration data left from previous applications.

2. **Configure:** The required Device channels and capabilities are enabled by configuration functions. Transmitter operation is enabled by creating a transmit Schedule. Filter Tables are configured for receiver operation. Special features of the Device may require extra configuration. Other low-level options such as Event Log List entries and bus parameters are also set at this point.
3. **Initialize data:** Data associated with transmit messages should be initialized before the Device is activated. Initialization prevents the transmission of invalid messages.
4. **Activate:** **`BTICard_CardStart`** activates all configured channels simultaneously. Once activated, the Device transmits and/or receives from the databuses independently of the host computer.
5. **Handle data:** The Device transmits and receives messages according to its configuration without requiring any host supervision. A program running on the host can update data for transmission or read received data at any time. To reduce data latency and/or host overhead, applications may access data in response to bus events detected by polling or interrupts. Driver functions are provided to simplify data exchange between the host and the Device.
6. **Deactivate:** **`BTICard_CardStop`** deactivates the Device. Unless the Device is explicitly deactivated, it continues operating even if the application software halts.
7. **Close:** Whether or not the Device is deactivated, **`BTICard_CardClose`** must be called before an application terminates. Failure to do so can cause unpredictable results. **`BTICard_CardClose`** does not deactivate the Device.

The following examples show how easy it is to perform these steps using the BTIDriver functions. Source code and an executable version of each of these examples is available on the distribution disk.

3.4 Transmit Example

In this section, we describe an example program that transmits the current frequency as a continuous message, and transmits remote tuning commands as a burst message (see Table 3.2). A sync message block is also transmitted in every frame. This example does not necessarily represent a real system but is useful for demonstrating how to transmit different message types. CSDB sources transmit messages at a repetition rate described by the CSDB specification. BTIDriver functions handle the timing requirements automatically based on the message update rates and the bus parameters of the channel.

To illustrate, recall that this example is simulating the transmission of continuous and burst messages with update rates shown in Table 3.2. The code in Figure 3.2 configures the Device to transmit these messages with proper timing.

The code starts by creating the message address array and the frequency array to be used for building the schedule. A byte array is also created to temporarily store transmit data.

```
HCARD hCard;
HCORE hCore;
INT corenum = 0;
INT cardnum = 0;
MSGADDR msgaddr[3];
INT   freq[3];
BYTE bufcont[6];

BTICard_CardOpen(&hCard,cardnum);                          //Open resources
BTICard_CoreOpen(&hCore,corenum,hCard);
BTICard_CardReset(hCore);
                                                          //Configure channel
BTICSDB_ChConfig(CHCFGCSDB_DEFAULT,BITRATECSDB_LOWSPEED,6,10,18,CH12,hCore);

                                                          //Create 3 messages
msgaddr[0] = BTICSDB_MsgCreate(MSGCRTCSDB_DEFAULT|MSGCRTCSDB_WIPESYNC,hCore);
msgaddr[1] = BTICSDB_MsgCreate(MSGCRTCSDB_DEFAULT,hCore);
msgaddr[2] = BTICSDB_MsgCreate(MSGCRTCSDB_DEFAULT|MSGCRTCSDB_BURST,hCore);

freq[0] = 1;
freq[1] = 2;
freq[2] = 1;

BTICSDB_SchedBuild(3,msgaddr,freq,0,0,CH12,hCore);       //Build a schedule

                                                 //Initialize continuous message
bufcont[0] = 0x12; //address byte
bufcont[1] = 0x80; //status byte
bufcont[2] = 0x00; //data = 121.1MHz
bufcont[3] = 0x10; // ""
bufcont[4] = 0x21; // ""
bufcont[5] = 0x00; //pad
BTICSDB_MsgDataWr(bufcont,6,msgaddr[1],hCore);

                                                  //Initialize burst message
BTICSDB_MsgDataByteWr(0x31,0,msgaddr[2],hCore); //address byte
BTICSDB_MsgDataByteWr(0xB1,1,msgaddr[2],hCore); //status byte
BTICSDB_MsgDataByteWr(0x90,3,msgaddr[2],hCore); //tune = 118.9MHz
BTICSDB_MsgDataByteWr(0x18,4,msgaddr[2],hCore); // " "

BTICard_CardStart(hCore);                                  //Start the card

while(!done)                  //Update data as required by the application
{
   if (update continuous message)
   {
      BTICSDB_MsgDataWr(data,6,msgaddr[1],hCore);
   }

   if (trigger burst message)
   {
      BTICSDB_MsgDataWr(data,6,msgaddr[2],hCore);
      BTICSDB_MsgValidSet(msgaddr[2],hCore);        //trigger burst message
   }
}

BTICard_CardStop(hCore);                                    //Stop the card
BTICard_CardClose(hCard);                                  //Close resources
```

Figure 3.2—Example transmit program

Like all OmniBus programs, the example in Figure 3.2 starts by opening the card and core. These provide handles for other functions. Concepts relating to cards, cores, and handles are described in detail in Appendix B, and other programming

manuals. It is good programming practice to reset the core, which puts it in a known state.

BTICSDB_ChConfig enables or disables selected options for a particular channel. Constants can be included to turn on specific options. Note that this function does not activate the channel. CH12 is a predefined constant referring to physical channel 12 of the Device. The channel is configured according to the CSDB bus parameters shown in Table 3.1: low speed bit rate of 12.5 Kbps, 6 bytes per block, 10 frames per second, and 18 blocks per frame.

The easiest way to create a Schedule that transmits the three messages at their proper update rates is to use the **BTICSDB_SchedBuild** function. To do this, we first create the three messages using **BTICSDB_MsgCreate**, and set the message address array equal to the returned addresses. We initialize the frequency array to reflect the update rate of the messages on the bus. Information on a given message is contained in the same position in each of these arrays. To determine the proper frequency value, divide the frames per second from the CSDB bus definition by the updates per second from the CSDB message definitions.

The **BTICSDB_SchedBuild** function constructs a Schedule in the Device memory that accurately maintains the message frequencies. A Schedule is a sequence of messages separated by timed gaps. The gaps are calculated so that the timing requirements of all messages are satisfied. The parameters in the example indicate that **BTICSDB_SchedBuild** is to schedule transmission of three messages on CH12 from the information in the arrays. There is to be zero gap between message blocks and zero reserved message blocks as the Schedule is built. **BTICSDB_SchedBuild** also uses the properties of the channel as configured with **BTICSDB_ChConfig**. An alternative to using **BTICSDB_SchedBuild** is to explicitly define your own Schedule as described in Section 4.4.

The Message Records can be initialized by using **BTICSDB_MsgDataWr** and/or **BTICSDB_MsgDataByteWr** before **BTICard_CardStart** commands the Device to begin transmitting. **BTICSDB_MsgDataWr** requires a temporary data buffer and then writes the entire buffer to the Message Record. **BTICSDB_MsgDataByteWr** writes one byte in a message block by specifying an index into the data array of the Message Record. Here we illustrate the initialization of the continuous message using a temporary data buffer and **BTICSDB_MsgDataWr**, and we initialize four bytes of the burst message using **BTICSDB_MsgDataByteWr**. Notice that we only write 4 of the 6 data bytes associated with the Message Record because when we created the message the data was set to 00h. The sync message was automatically initialized with sync characters, A5h, by **BTICSDB_MsgCreate** with the MSGCRTCSDB_-WIPESYNC flag.

Once the device is started with **BTICSDB_CardStart** the messages are transmitted at the proper rates until the Device is halted by **BTICard_-CardStop**. While the Device is running, message data may be changed as necessary. The burst message is transmitted in response to **BTICSDB_MsgValidSet**. As always, the program ends with

BTICard_CardClose to free up the resources for other applications. Transmitting messages on other transmit channels only requires extra **BTICSDB_ChConfig**, **BTICSDB_MsgCreate**, and **BTICSDB_Sched-Build** function calls.

3.5 Receiver Example

In this section, we describe an example program that receives the same messages that were created in the previous transmitter example. Specifically, we will configure the Device to receive the Current Frequency and Remote Tuning messages. The primary task in configuring receive channels is telling the Device to which Message Records it should store specific messages. The Device ignores messages not assigned a Message Record unless a default record has been defined. If a default record is defined, all messages not assigned their own Message Records are written to the default record. The default record makes it possible to receive all bus traffic while isolating the messages of interest.

The code in Figure 3.3 configures the Device to receive all traffic. The two messages of interest are stored in separate Message Records. The Device writes all other messages, including the sync message, to the default record.

```
HCARD hCard;
HCORE hCore;
INT cardnum = 0;
INT corenum = 0;
MSGADDR msgdefault;
MSGADDR msgcont;
MSGADDR msgburst;
BYTE datadefault[6];
BYTE datacont[6];
BYTE databurst[6];

BTICard_CardOpen(&hCard,cardnum);                          //Open resources
BTICard_CoreOpen(&hCore,corenum,hCard);
BTICard_CardReset(hCore);

                                                        //Configure channel
BTICSDB_ChConfig(CHCFGCSDB_DEFAULT,BITRATECSDB_LOWSPEED,6,10,18,CH4,hCore);

                                                          //Define filters
msgdefault = BTICSDB_FilterDefault(MSGCRTCSDB_DEFAULT,CH4,hCore);
msgcont = BTICSDB_FilterSet(MSGCRTCSDB_DEFAULT,0x12,SIALL,CH4,hCore);
msgburst = BTICSDB_FilterSet(MSGCRTCSDB_DEFAULT,0x31,SIALL,CH4,hCore);

BTICard_CardStart(hCore);

while(!done)
{
                                                      //Read message records
   BTICSDB_MsgDataRd(datadefault,6,msgdefault,hCore);
   BTICSDB_MsgDataRd(datacont,6,msgcont,hCore);
   BTICSDB_MsgDataRd(databurst,6,msgburst,hCore);

   //process data as required by the application
}

BTICard_CardStop(hCore);
BTICard_CardClose(hCard);                                  //Close resources
```

Figure 3.3—Example receiver program

This example starts out similar to the previous one. We first define message addresses named *msgdefault*, *msgcont*, and *msgburst*. Byte arrays are created to

store the data read from Message Records. As always, the card and core are opened using **BTICard_CardOpen** and **BTICard_CoreOpen**. It is good programming practice to call **BTICard_CardReset** to put the Device into a known state.

BTICSDB_ChConfig configures the channel with the bus parameters provided from Table 3.1. The function sets up an empty Filter Table for a receive channel and then configures the channel with default options.

A Filter Table is an array of pointers to Message Records. There is one pointer for every possible message type (i.e., every possible address/source identifier combination). All pointers are initially zero. A received message is recorded only if its entry in the Filter Table points to a valid Message Record. The **BTICSDB_FilterDefault** function fills the Filter Table with pointers so all messages received on that channel are written to the default record. If **BTICSDB_FilterDefault** is used, it must precede any calls to **BTICSDB_FilterSet** or **BTICSDB_FilterWr**. The two calls to **BTICSDB_FilterSet** assign receive messages to Message Records by placing entries in the Filter Table. The **BTICSDB_FilterDefault** and **BTICSDB_FilterSet** functions return the address of the Message Record.

The SIALL constant tells the Device to accept all messages with the given address and any source identifier (see Section 4.3). Different constants may be used to specify that only messages with specific SIs are to be received. Messages with different SIs could be assigned to separate Message Records.

The Device begins receiving messages only after **BTICard_CardStart** is called. As discussed above, when a message is received, it is stored in its assigned Message Record. The previous value in that Message Record is overwritten. **BTICSDB_MsgDataRd** may be called at any time to return the message data array from a specified Message Record. The data will be written to the buffer supplied by the user. The user can parse the array of bytes and extract the address, pertinent information from the status byte, and the message specific data bytes.

The program ends with the required **BTICard_CardStop** and **BTICard_CardClose** functions. Receiving messages on other receive channels only requires extra calls to **BTICSDB_ChConfig**, **BTICSDB_FilterDefault**, and **BTICSDB_FilterSet**.

3.6 Monitor Example

In a sense, any receiver is a monitor that holds the most recent value of the received data. However, Ballard Devices have a Sequential Monitor that records a time-tagged history of user-selected bus activity in what is called a Sequential Record. A Sequential Record is useful for analyzing and reconstructing all or selected bus activity. Additional information on the Sequential Record may be found in Section 4.6.

The code in Figure 3.4 combines aspects of the previous receive and transmit examples while configuring the Sequential Record to capture only the message types defined in the examples. Specifically, the Sequential Record will capture

only the sync and continuous messages from the receive channel and all messages transmitted by the Device. Note that the burst message is not included in this example.

Filtering irrelevant messages out of the bus traffic conserves Sequential Record memory and makes the Sequential Record easier to analyze. Filtering for the Sequential Record can be established at either the channel level or the message level. If the CHCFGCSDB_SEQALL constant is used in **BTICSDB_ChConfig**, then all CSDB messages on that channel are saved in the Sequential Record. If only specific messages are to be saved then the CHCFG_SEQSEL constant is used in **BTICSDB_ChConfig**, and the MSGCRTCSDB_SEQ constant is used in either **BTICSDB_FilterSet** for receive or **BTICSDB_MsgCreate** for transmit. These are summarized in Table 3.3.

Use these functions and parameters → to record ↓	BTICSDB_ChConfig	Receive BTICSDB_FilterSet	Transmit BTICSDB_MsgCreate
from all CSDB messages on selected channels	CHCFGCSDB_SEQALL	don't care	don't care
from selected CSDB messages on selected channels	CHCFGCSDB_SEQSEL	MSGCRTCSDB_SEQ	MSGCRTCSDB_SEQ

Table 3.3—Sequential Record filtering options

Much of the code in Figure 3.4 is a combination of modified fragments from the previous examples. Since we want to monitor all transmitted messages, the CHCFGCSDB_SEQALL constant is used in place of the CHCFGCSDB_DEFAULT constant in **BTICSDB_ChConfig** for CH12. We only want to save filtered messages on the receive channel so the CHCFGCSDB_SEQSEL constant is used in **BTICSDB_ChConfig** for CH4. To save the sync and continuous messages in the Sequential Record, the MSGCRTCSDB_SEQ constant is used in their respective **BTICSDB_FilterSet** functions. **BTICard_SeqConfig** allocates memory for and configures the Sequential Record. The SEQCFG_DEFAULT constant selects all default settings. **BTICard_CardStart** activates all configured channels on the Device. After activation, the Device begins simultaneously receiving, transmitting, and recording the specified traffic in its Sequential Record.

The while() loop in the code polls and processes the Sequential Record. For illustration purposes, this example simply prints each channel number, CSDB message address, byte count, and time-tag. **BTICard_SeqBlkRd** copies the available records from the on-board Sequential Record to the user-supplied buffer (seqbuf) and returns the number of words copied (seqcount). To process records in seqbuf, it is necessary to find the start and type of each record, which may be done with different **BTICard_SeqFind??** functions. Here, **BTICard_SeqFindInit** initializes a structure (sfinfo). Then, repeated calls to **BTICard_SeqFindNextCSDB** find and point (pRecCSDB) to the next occurrence of a CSDB-type record. The desired values are then printed from the record. The Sequential Record may hold messages from multiple protocols if the device supports multiple protocols on the same core. If the application requires it, the received and transmitted messages may be read/written as in the previous examples. As usual, the program ends with **BTICard_CardStop** and **BTICard_CardClose**.

```
HCARD hCard;
HCORE hCore;
INT corenum = 0;
INT cardnum = 0;
MSGADDR msgaddr[2];
INT freq[2];
BYTE bufcont[6];
USHORT seqbuf[2048];
ULONG seqcount;
ULONG blkcnt;
LPSEQRECORDCSDB pRecCSDB;
SEQFINDINFO sfinfo;

BTICard_CardOpen(&hCard,cardnum);
BTICard_CoreOpen(&hCore,corenum,hCard);
BTICard_CardReset(hCore);

                                                        //Configure channels
BTICSDB_ChConfig(CHCFGCSDB_SEQSEL,BITRATECSDB_LOWSPEED,6,10,18,CH4,hCore);

BTICSDB_ChConfig(CHCFGCSDB_SEQALL,BITRATECSDB_LOWSPEED,6,10,18,CH12,hCore);

                                                         //Create 2 messages
msgaddr[0] = BTICSDB_MsgCreate(MSGCRTCSDB_WIPESYNC,hCore);
msgaddr[1] = BTICSDB_MsgCreate(MSGCRTCSDB_DEFAULT,hCore);

freq[0] = 1;                                  //Set up transmit update rates
freq[1] = 2;

BTICSDB_SchedBuild(2,msgaddr,freq,0,0,CH12,hCore);       //Build a schedule


bufcont[0] = 0x12; //address byte
bufcont[1] = 0x80; //status byte
bufcont[2] = 0x00; //data = 121.1MHz
bufcont[3] = 0x10; // ""
bufcont[4] = 0x21; // ""
bufcont[5] = 0x00; //pad
BTICSDB_MsgDataWr(bufcont,6,msgaddr[1],hCore);

                               //Define sync and continuous message filters
BTICSDB_FilterSet(MSGCRTCSDB_SEQ,0xA5,SIALL,CH4,hCore);
BTICSDB_FilterSet(MSGCRTCSDB_SEQ,0x12,SIALL,CH4,hCore);

BTICard_SeqConfig(SEQCFG_DEFAULT,hCore); //Configure the sequential monitor

BTICard_CardStart(hCore);

while(!done)
{                                               //Read the sequential record
   seqcount = BTICard_SeqBlkRd(seqbuf,2048,&blkcnt,hCore);
   BTICard_SeqFindInit(seqbuf,seqcount,&sfinfo); //Initialize Find function
   while(!BTICard_SeqFindNextCSDB(&pRecCSDB,&sfinfo))   //Find CSDB records
   {
      //Write to disk, display data, etc. as desired. For example:

      printf("\nCh:%02u  Addr:%08lX  Bytes:%d  Time:%08lX",
            ((pRecCSDB->activity & MSGACTCSDB_CHMASK) >> MSGACTCSDB_CHSHIFT),
            pRecCSDB->data[0],
            pRecCSDB->datacount,
            pRecCSDB->timestamp);
   }

   //Also, read and write message data as desired.

}

BTICard_CardStop(hCore);
BTICard_CardClose(hCard);
```

Figure 3.4—Example monitor program

The preceding examples illustrated the most important BTIDriver functions. Your programs for your Device can be modeled after the code in these examples. Complete source code for these examples and more may be found on the distribution disk. Detailed function descriptions are found in Appendix A. More information may be contained in the README.TXT files on the distribution disk.

This page intentionally blank.

4. ADVANCED OPERATION

The purpose of this chapter is to provide you with a deeper understanding of how your Device works internally. Such insight will help you use the BTIDriver functions for more advanced applications. This chapter describes how the capabilities of the Device are implemented in its internal data structures. Some operational details omitted from the previous chapter are also included.

4.1 Overview

A Digital Signal Processor (DSP) is the heart of the Device. Resident firmware executed by the DSP implements many of the Device capabilities. The speed of the DSP allows simultaneous operation of all CSDB channels. Gate arrays manage the interface to the host, arbitrate access to the on-board memory, and provide CSDB encoding and decoding. The on-board memory contains all configuration data structures described in the following sections.

All exchanges between the host and CSDB databuses are buffered by various data structures in the Device memory. The host writes messages for transmission to designated data structures, and the Device transmits them according to its configuration. Similarly, the Device receives messages from the CSDB databus(es) and stores them in its memory. The host can then read the received messages from designated locations. User software accesses these structures through calls to driver functions.

Figure 4.1 provides an overview of the flow of messages between the primary data structures in the Device. These data structures and their interactions are explained in detail in following sections. It will be helpful to refer back to this diagram.

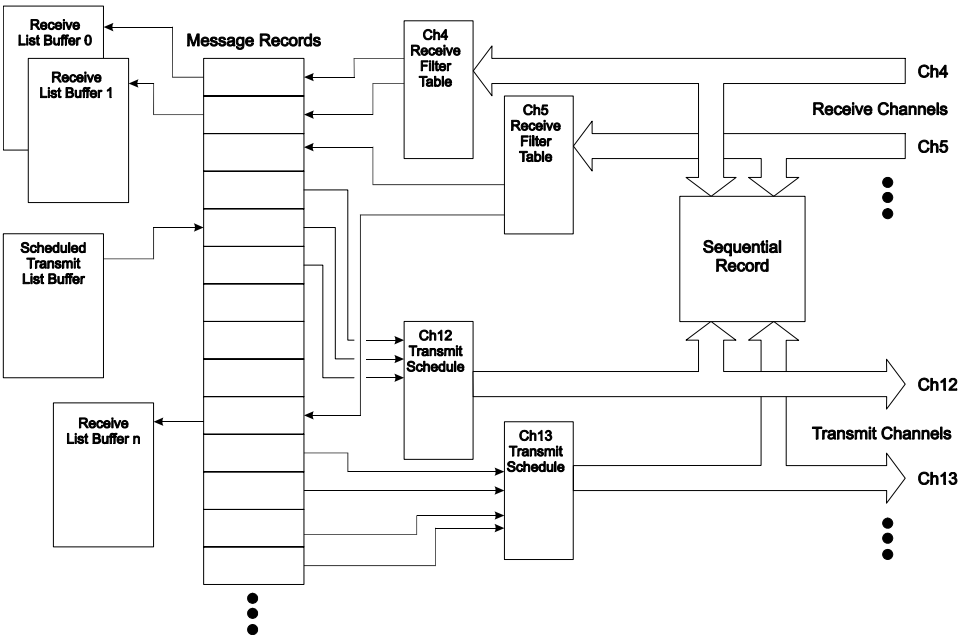


Figure 4.1—Message flow between primary data structures in a CSDB Device

4.2 Message Records

Message Records are used by both receive and transmit channels to buffer incoming and outgoing messages. This was illustrated in the example programs in Chapter 3. A Message Record structure (Figure 4.2) includes space for additional information that may be used if the corresponding options are enabled when the Device is configured. By default, only the Reserved, Activity, Data Count, and Data fields are used. The Device updates the fields continuously while it is activated (i.e., between **`BTICard_CardStart`** and **`BTICard_CardStop`**). (Note that some options are mutually exclusive.) You can access the Message Record by calling **`BTICSDB_MsgBlockRd`** and **`BTICSDB_MsgBlockWr`**. The rest of this section describes the Message Record fields. Sections 4.3 and 4.4 describe how the receive and transmit channels use Message Records.

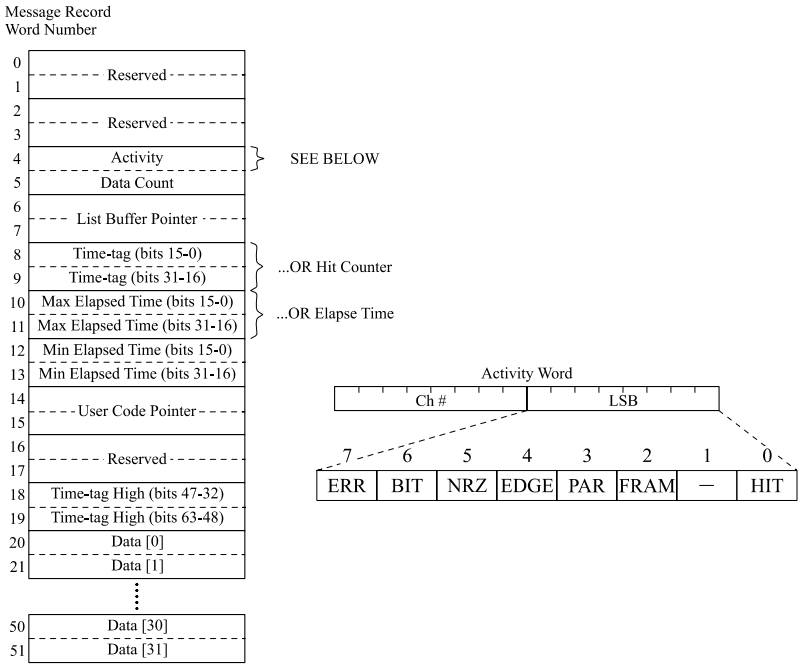


Figure 4.2—Message Record structure

4.2.1 Reserved

The reserved words may be used by internal processes and are not intended for end users.

4.2.2 Activity

The Activity word provides a variety of useful information about the message. The following describes the fields within the Activity word:

Ch #: The channel number is an eight-bit field identifying the source or destination channel of a message according to the Device channel numbering. The DSP senses the channel and automatically fills in this value, which is in the range of 0 to 255. To evaluate the channel number, the activity field can be logically ORed with the MSGACTCSDB_CHMASK flag, then shifted by MSGACTCSDB_CHSHIFT to the right.

ERR: This bit indicates an error was detected in one of the bytes in the message, and only has meaning for received messages. It is the logical OR of BIT, NRZ, EDGE, PAR, and FRAME.

BIT: This bit indicates a bit error was detected in one of the bytes in the message, and only has meaning for received messages. It is the logical OR of NRZ and EDGE.

NRZ: Indicates that one of the bytes in the message was received with a NRZ error. This bit only has meaning for received messages. An NRZ error may indicate poor cabling or signal integrity.

EDGE: Indicates that one of the bytes in the message was received with an edge error on a bit. This bit only has meaning for received messages. An edge error may indicate poor cabling or signal integrity.

PAR: Indicates that one of the bytes in the message was received with a parity error. This bit only has meaning for received messages.

FRAME: Indicates that one of the bytes in the message was received with a framing error. This bit only has meaning for received messages.

HIT: Indicates that the Message Record has been processed by either transmission or reception of a message, so this bit is normally set when the Device is operating. The **BTICSDB_MsgIsAccessed** function returns the value of the Hit bit and then clears it. When the Hit bit is set, the user knows the message has been processed at least once since the previous call to **BTICSDB_MsgIsAccessed**. This bit has meaning for both received and transmitted messages.

4.2.3 Data Count

The Data Count is a 16-bit value indicating the number of valid data words in the corresponding Data field of the Message Record. This value may be used to specify the number of bytes to be read with **BTICSDB_MsgDataRd**. Typically a user will not need to use the Data Count because the number of bytes per block is defined for every channel using **BTICSDB_ChConfig**.

4.2.4 List Buffer pointer

When a List Buffer is associated with a Message Record, this field holds a pointer to the List Buffer. See Section 4.7 for more information on List Buffers.

4.2.5 Time-tag

The time-tag is a 32-bit value derived from an internal clock. The resolution (and resulting range) of time-tag values may be adjusted with the **BTICard_TimerResolution** function. This function does not affect the internal Device timer. It only determines which bits are extracted from the Device timer to form the 32-bit time-tag. The selected resolution applies to all channels. The Time-tag field alternately holds the lower 32 bits of the 64-bit IRIG time-tag if the timer is configured as an IRIG timer using **BTICard_IRIGConfig**.

Note that the time-tag of a transmitted message represents when the message is loaded into the encoder, *not* when the message is actually transmitted. Similarly, the time-tag of a received message represents when the message was read from the decoder. Thus, the time-tag can deviate slightly from the actual time of the CSDB bus activity.

4.2.6 Hit Counter

This number is incremented every time the Message Record is accessed by the DSP, so it represents the number of times a message has been received or transmitted. This option is enabled for all messages on the channel by **`BTICSDB_ChConfig`** or for a single message by **`BTICSDB_MsgCreate`**, **`BTICSDB_FilterDefault`**, and **`BTICSDB_FilterSet`**. Since the Hit Counter uses the same field as the time-tag, the Hit Counter may not be used concurrently with any of the time-related fields.

4.2.7 Min/Max Elapsed Time

The Minimum and Maximum Elapsed Times are the worst case Elapsed Times since the Device was last activated. The DSP calculates the Elapsed Time, and if it is not between the current minimum and maximum, the appropriate value is updated. The Min/Max Elapsed Time options are enabled by the **`BTICSDB_ChConfig`** function. This function also initializes the minimum time and the maximum time to zero.

4.2.8 Elapsed Time

The Elapsed Time is the most recent transmit interval (i.e., the time between the two most recent receptions or transmissions of a particular message.) The DSP calculates this value by subtracting the previous time-tag from the current time-tag. Resolution of the Elapsed Time is the same as the time-tag. The Elapsed Time option is enabled by the **`BTICSDB_ChConfig`** function. Since the Elapsed Time uses the same field as the Maximum Elapsed Time, they may not be used concurrently.

4.2.9 User code pointer

The user can specify individual Message Records to be serviced by custom DSP code. Contact Ballard Technology for further information on use of custom DSP code.

4.2.10 Time-tag High

The Time-tag High field holds the upper 32 bits of the 64-bit IRIG time-tag if the IRIG timer is enabled.

4.2.11 Data

The Data field is an array of 32 16-bit data values. Only the lower 8 bits of each element in the array are used for data storage; the upper 8 bits are reserved. When a transmit message is created with **`BTICSDB_MsgCreate`** the Data array is initialized to zeroes, ones, or A5h depending on the configuration options selected. When a message is received, each byte of the message block is stored in

an individual element of the Data array and the total number of bytes stored in the Data array is written to the Data Count field.

4.3 Filter Tables

Each receive channel has a Filter Table, as shown in Figure 4.3. A Filter Table is an array of pointers to Message Records. It contains one pointer for every address/SI combination (256 x 8 pointers). The address byte and the selected SI bits of the status byte of the CSDB message block form an index into the Filter Table. A filter can be configured to examine any combination of up to 3 SI bits in the status byte. The SI bits are usually bit 0 and bit 1 of the status byte but sometimes include bit 3 of the status byte.

4.3.1 How Filter Tables work

When a message is received from the CSDB databus, the Device examines the address and SI bits and retrieves the corresponding pointer from the Filter Table. If the pointer is zero, the Device quits processing the message. Otherwise, the Device writes the message to the Message Record to which the Filter Table points for that address/SI. Other processing may follow depending on which options are enabled. For example, if the Elapsed Time option is enabled, the Device calculates the time elapsed since the last reception of the word and writes this to the Message Record.

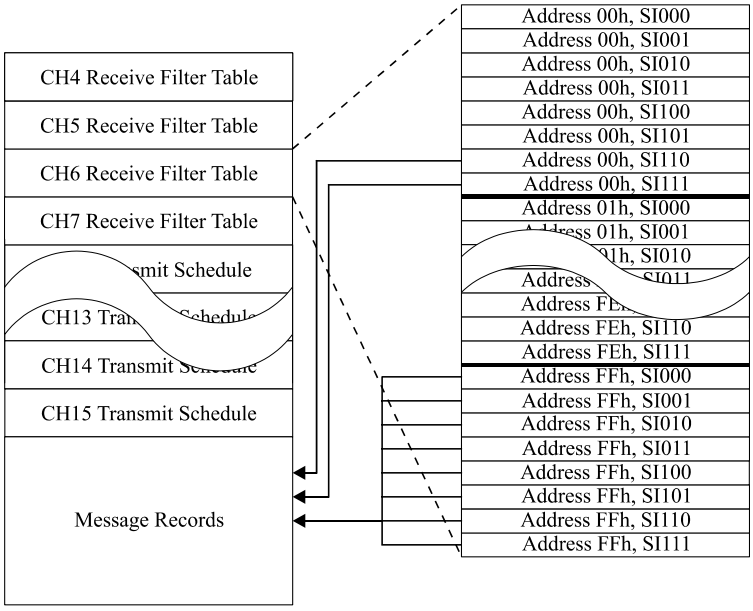


Figure 4.3—Filter Tables

4.3.2 Configuring the Filter Tables

The user sets Filter Table entries to point to Message Records with the **`BTICSDB_FilterSet`** and **`BTICSDB_FilterDefault`** functions. **`BTICSDB_ChConfig`** sets up the Filter Table and therefore must precede any calls to **`BTICSDB_FilterSet`** or **`BTICSDB_FilterDefault`**. Any number of

Filter Table entries may point to the same Message Record. For example, the SIALL constant used in the **`BTICSDB_FilterSet`** function sets the pointers for all eight SIs of a particular address to the same Message Record, as shown for address 0xFF in Figure 4.3.

4.4 Transmit Schedules

A Schedule controls the transmission of messages onto the CSDB databus and is executed by the Device firmware. There are two ways to create a Schedule. Chapter 3 demonstrated the easiest way: using the **`BTICSDB_SchedBuild`** function to automatically construct a Schedule. This section describes the Schedule in more detail and tells how to explicitly construct a Schedule.

4.4.1 How Schedules work

The Schedule must be loaded by the host computer into the Device's on-board memory as a series of Command Blocks. Typically, each transmit channel is allocated 512 Command Blocks as shown in Figure 4.4. Each Command Block contains one of the opcodes shown in Table 4.1.

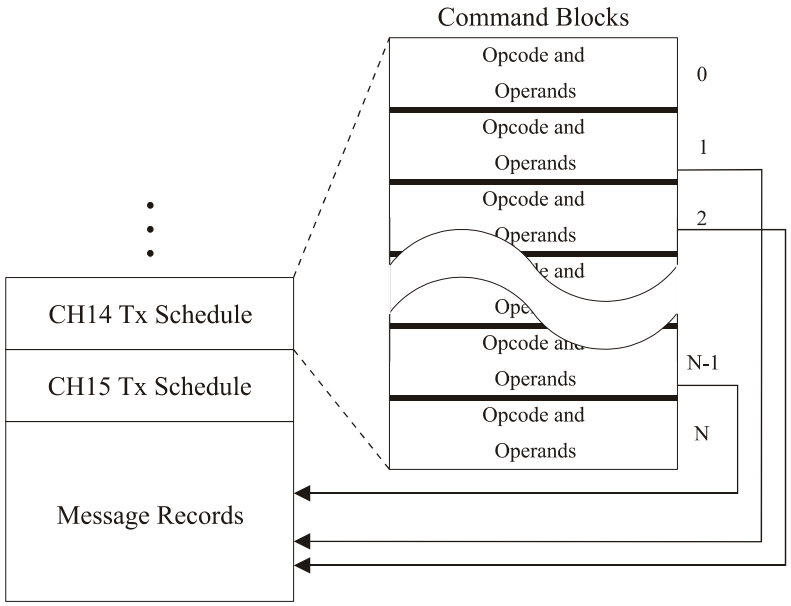


Figure 4.4—Transmit Schedule

Opcode Name	Function	Description
MESSAGE	BTICSDB_SchedMsg	Transmits the CSDB message from the Message Record
GAP	BTICSDB_SchedGap	Inserts a timed gap between transmissions
ENTRY	BTICSDB_SchedEntry	Indicates the starting point for the Schedule
HALT	BTICSDB_SchedHalt	Halts the Schedule
PAUSE	BTICSDB_SchedPause	Halts processing of Schedule until BTICSDB_ChResume is called
RESTART	BTICSDB_SchedRestart	Restarts the Schedule at the beginning
LOG	BTICSDB_SchedLog	Generates Event Log List entry (see Section 4.8.1)
USERCODE	BTICSDB_SchedUser	Calls custom DSP code
BRANCH	BTICSDB_SchedBranch	Jumps to specified Command Block and resumes execution.
CALL	BTICSDB_SchedCall	Jumps to specified Cmd Block, saving the return address on stack.
RETURN	BTICSDB_SchedReturn	Returns to address following last call.
NOP	BTICSDB_SchedNop	No operation.
PULSE	BTICSDB_SchedPulse	Pulse an external discrete signal

Table 4.1—Command Blocks in the transmit Schedule

Though very complex Schedules may be created using the special commands listed in Table 4.1, the typical Schedule consists of a loop of MESSAGE and GAP Command Blocks. When the DSP processes a MESSAGE Command Block, it retrieves the CSDB message to be transmitted from the Message Record pointed to by the operand in the Command Block (see Figure 4.4). The DSP loads the data into the encoder and may update fields of the Message Record, depending on which options are enabled for that message. It then proceeds to the next Command Block.

A GAP Command Block specifies the period the transmitter must wait before starting another transmission. The DSP services other channels during transmissions and gap times.

A Schedule is required for any transmission from the Device. Even if the Device is to transmit only one message one time, a Schedule must be created. To transmit one message at a time, the Schedule would consist of a MESSAGE Command Block followed by a HALT Command Block.

A few rules must be followed when developing a Transmit Schedule:

1. The parameter in a GAP Command Block is the total time from the end of one message to the start of the next messages. The gap is measured in bit times at the speed set for that channel. For long gap times, GAP Command Blocks may be strung together.

2. There is an implicit RESTART at the end of every Schedule, so the Schedule runs in a loop unless directed otherwise (e.g., by a HALT Command Block).

3. A subroutine (the destination for any CALL or BRANCH) should precede the use of the CALL or BRANCH in the Schedule. The main starting point should follow subroutines and is indicated by the ENTRY Command Block.

4.4.2 Creating a Schedule

The Schedule is programmed explicitly using **`BTICSDB_SchedMsg`**, **`BTICSDB_SchedGap`**, and other scheduling functions shown in Table 4.1.

Each of these functions makes an entry into the next available Command Block at the end of the current Schedule. The opcode in the Command Block corresponds to the name of the function, and the operand is specified in the function parameters. Only the MESSAGE and GAP opcodes specifically relate to the transmission of CSDB messages. The other opcodes controlling the processing of the Schedule are explained for the individual functions in Appendix A.

4.5 General-Purpose Serial

An OmniBus CSDB Device can be configured to transmit and receive general-purpose serial data using the procedures outlined below. An example of general-purpose serial communication can be found on the distribution disk.

4.5.1 Transmit Procedure

1. **Configure:** Use the CHCFGCSDB_ASYNC constant as well as any parity option constants for **`BTICSDB_ChConfig`**. Set the number of bytes per block to one. Set the bit rate to suit your application. If you use **`BTICSDB_SchedBuild`** in step 3, configure the number of blocks per frame and frames per second to suit your application.
2. **Create Message:** Create a non-continuous message using the MSGCRTCSDB_NONCONT flag for **`BTICSDB_MsgCreate`**.
3. **Schedule:** Schedule the non-continuous message automatically using **`BTICSDB_SchedBuild`** or explicitly using **`BTICSDB_SchedMsg`**.
4. **Create a List:** Associate a List Buffer with the scheduled transmit message using **`BTICSDB_ListXmtCreate`** (see Section 4.7).
5. **Write Data:** Data can be written to the list buffer using **`BTICSDB_ListDataWr`**. The non-continuous message must be triggered using **`BTICSDB_MsgValidSet`**. Each time the list reaches an empty state, the message must be triggered again.

4.5.2 Receive Procedure

1. **Configure:** Use the CHCFGCSDB_ASYNC constant as well as any parity option constants for **`BTICSDB_ChConfig`**. Set the number of bytes per block to one. Set the bit rate to suit your application. Asynchronous receive channels ignore the frames per second and blocks per frame parameters for **`BTICSDB_ChConfig`**.
2. **Create a Filter:** Create a default filter with the SIALL constant for **`BTICSDB_FilterDefault`** to receive all data values.
3. **Create a List:** Associate a List Buffer with the default filter using **`BTICSDB_ListRcvCreate`** (see Section 4.7).
4. **Read Data:** Data can be read from the list buffer using **`BTICSDB_ListDataRd`**.

4.6 Sequential Record

The Sequential Monitor records a time-tagged history of selected CSDB messages transmitted and received by the Device. This history is stored in a buffer called the Sequential Record. The Sequential Record can store messages from any or all Device channels, including other protocols if supported by the

Device (see Appendix B for more information). Individual channels and/or individual messages within a channel may be selectively recorded. The filtering of desired messages is controlled as described in Section 3.6 and Table 3.3.

By default, recording halts when the on-board Sequential Record is full. This happens to prevent unread data from being overwritten when the host gets behind in reading data from Sequential Record. However, if the Sequential Record is configured with the SEQCFG_CONTINUOUS flag in **`BTICard_SeqConfig`**, recording is not halted. In this mode, it automatically wraps around and continues recording, overwriting old messages. The Device can log an entry in the Event Log List when the Sequential Record either halts or wraps around (depending on the selected option). Alternatively, the Device can log an entry on every n^{th} message recorded into the Sequential Record. (See Section 4.8 for more information on enabling entries into the Event Log List.)

The Sequential Monitor starts recording data to the Sequential Record with **`BTICard_CardStart`**. The Device must be running (**`BTICard_CardStart`**) for data to be recorded in the Sequential Record. However, while the Device is running, the Sequential Record can be stopped and restarted without affecting other Device functions. **`BTICard_SeqStart`** is used to start the Sequential Record; it also stops and initializes it if necessary before restarting it. **`BTICard_SeqStop`** stops data from being added to the Sequential Record. If **`BTICard_SeqResume`** is called after **`BTICard_SeqStop`**, data recording continues and the original data is not lost.

Head and tail pointers are used to keep track of the location of the most recently entered data and the oldest data that needs to be read. When the DSP adds a message to the Sequential Record, it updates the head pointer; when the host reads the contents of the Sequential Record using **`BTICard_SeqRd`**, the tail pointer is updated. As long as the **`BTICard_SeqRd`** function is used to read the Sequential Record, the user need not maintain the head and tail pointers.

Two additional options control the contents of the Sequential Record: Interval mode and Delta mode. In Interval mode, the Device only records the first instance of a message in successive intervals of time (as illustrated in Figure 4.5). In Delta mode, an entry is added to the Sequential Record for a message only when the data changes. Both options are enabled by including the proper flags in the **`BTICard_SeqConfig`** function.

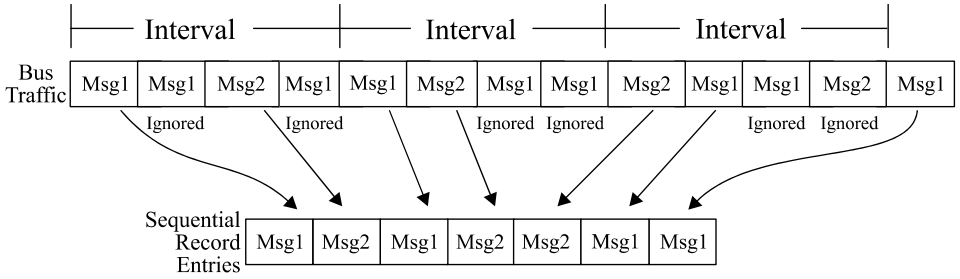


Figure 4.5—Operation of Sequential Record in Interval mode

4.7 List Buffers

By providing a separate location for every address and SI combination, the Message Record array simplifies retrieving received data and updating transmitted data. However, each Message Record holds only one CSDB message. Alternatively, List Buffers can be used to store sequences of messages. Lists Buffers may be used to buffer a changing receive message, to transmit a predefined sequence of messages, or to support general-purpose, buffered, serial communications.

A List Buffer may service messages on either a receive or a transmit channel. When a List Buffer is associated with a message, the List Buffer Pointer field in the Message Record (shown in Figure 4.2) holds a pointer to the List Buffer. The different types of List Buffers have the same structure, but operate very differently as described in the following sections. The message type (continuous, non-continuous, or burst) and channel type (transmit or receive) that the List Buffer is associated with affects the operation as well. Example programs that use list buffering may be found on the distribution disk.

List Buffers are defined by the user using the **`BTICSDB_ListRcvCreate`**, and **`BTICSDB_ListXmtCreate`** functions. The size of each List Buffer is measured in the number of CSDB messages it holds. The number of List Buffers that can be allocated and the size of each are limited only by available Device memory.

4.7.1 Receive List Buffers

There are two types of List Buffers (FIFO and ping-pong) that can be associated with a receive message. The type is specified using predefined constants when the list is created with **`BTICSDB_ListRcvCreate`**.

The Device adds messages to a List Buffer as they are received, and the user retrieves them with the **`BTICSDB_ListDataRd`** function. The user is not required to maintain the pointers as long as the **`BTICSDB_ListDataRd`** function is used to access the buffer. The pointers are automatically changed when a new message is entered into the list (i.e., when it is received) and when a message is read from the list using **`BTICSDB_ListDataRd`**.

FIFO List Buffer:

When the value of received data in a particular address/SI is rapidly changing and it is important not to lose any of the data, then a FIFO receive List Buffer may be used. If messages are not read from the receive List Buffer fast enough, the buffer wraps around and overwrites old messages. The Device can generate an Event Log List entry to signal this occurrence.

Ping-Pong List Buffer:

The ping-pong List Buffer guarantees data integrity by preventing a problem that can occur when only a single buffer is used. The problem happens when the host computer and the on-board processor simultaneously access the same message, causing the data being read by the host to contain part of the old message and part of the new message. The ping-pong List Buffer solves this problem by using multiple memory locations, so that **`BTICSDB_ListDataRd`** always reads the most recent complete copy of a received message.

4.7.2 Scheduled transmit List Buffers

There are three types of List Buffers (FIFO, ping-pong, and circular) that may be associated with scheduled transmit messages. A scheduled transmit List Buffer is attached to a Message Record and is created using **`BTICSDB_ListXmtCreate`**. The type of List Buffer is specified using predefined constants. The user writes messages sequentially to the List Buffer using the **`BTICSDB_ListDataWr`** function. The user is not required to maintain the pointers as long as the **`BTICSDB_ListDataWr`** function is used. Transmission of a message from a List Buffer depends on the message type (continuous, non-continuous, or burst) and the List Buffer type. Specific transmit behavior is discussed for each list buffer type and message type below.

FIFO List Buffer:

Whenever the Schedule indicates that a message should be transmitted from a Message Record associated with a FIFO List Buffer, the next available word is obtained from the FIFO List Buffer and transmitted by the Device.

> **Continuous:** If messages are not updated fast enough by the host and all messages have been transmitted at least once, then the last (most recent) message written by the host to the FIFO List Buffer is the message that is transmitted until another message is written by the host. The Device can generate an Event Log entry to signal that more data needs to be written by the host.
>
> **Non-continuous/Burst:** Non-continuous or burst Message Records associated with FIFO List Buffers must be set valid by calling **`BTICSDB_MsgValidSet`**. Once the message is set valid, the entire list is transmitted at the scheduled rate (one transmission for non-continuous messages and sixteen transmissions for burst messages) until it is empty, then the message is set invalid and is no longer transmitted. The Device can generate an Event Log entry to signal that the FIFO is empty and the message is no longer being transmitted in the schedule.

Ping-Pong List Buffer:

As in receive, the transmit ping-pong List Buffer guarantees data integrity when the host computer and the Device processor simultaneously access the same message, which could cause the data being transmitted by the host to contain part of the old message and part of the new message.

> **Continuous:** With a ping-pong List Buffer associated with a continuous message, the Device continuously transmits the last complete message loaded using **`BTICSDB_ListDataWr`** at the schedule rate.
>
> **Non-continuous/Burst:** A ping-pong List Buffer associated with a non-continuous or burst message transmits the last complete message loaded using **`BTICSDB_ListDataWr`** as long as the message was set valid with **`BTICSDB_MsgValidSet`**. By default, a non-continuous message is transmitted one time and then is no longer valid. A burst message is transmitted sixteen times and then is no longer valid.

Circular List Buffer:

With a circular List Buffer, transmissions repeatedly loop through the entire list buffer. This feature greatly simplifies the transmission of a data pattern, for example a sine wave or ramp, since the whole pattern could be preloaded into the List Buffer rather than requiring the host computer to update the data value for each transmission. The behavior of a circular List Buffer for non-continuous and

burst messages is undefined and therefore is only recommended for use with continuous messages.

4.8 Special Events

In some software programs, it may be necessary to know when a special event has happened. Examples of special events are when a specific message is received, when the transmit Schedule reaches a certain point, when an error occurs, or when the Sequential Record or a List Buffer needs service. An Event Log List is used to record these special events. Notification that the special event has occurred can happen through polling or interrupts. If your Device supports multiple protocols, the Event Log List can contain events from more than just CSDB related activity. See the description of **`BTICard_EventLogRd`** in Appendix A for a table of event types for CSDB. Refer to other BTIDriver Protocol manuals for event types for different protocols.

4.8.1 Event Log List

To use an Event Log List, it is necessary to create the Event Log List and to enable the specific events that are to be recorded. The **`BTICard_EventLog-Config`** function creates the Event Log List. To enable a specific event, use the corresponding enabling constant in the enabling function. Many BTIDriver functions can be configured to enable Event Log List entries. See the description of **`BTICard_EventLogRd`** in Appendix A for a table that lists the enabling function(s) for each event.

The Event Log List is a circular buffer, which records all events in order of occurrence. An entry is added to this list each time an enabled event occurs. An event is identified by reading and evaluating its entry from the Event Log List. Each Event Log List entry contains three fields. The description of **`BTICard_EventLogRd`** in Appendix A summarizes the meanings and values of these parameters.

4.8.2 Polling

When the program is running, the Event Log List may be polled using **`BTICard_EventLogRd`**. This function returns a zero if the Event Log List is empty. Otherwise, it may be evaluated to determine the source of the event. See the description of **`BTICard_EventLogRd`** in Appendix A for a table that describes the Event Log List fields. Each entry in the Event Log List may be read only once, since **`BTICard_EventLogRd`** automatically increments the list pointers each time it is called. The **`BTICard_EventLogRd`** function returns the oldest entry from this list and updates the tail pointer. See the polling example on the software distribution disk.

4.8.3 Interrupts

If your Device supports hardware interrupts, you can configure it to issue a hardware interrupt each time an entry is made in the Event Log List, which virtually becomes an interrupt log list.

Using hardware interrupts requires an interrupt service routine and an understanding of the computer's operating system. The **`BTICard_IntIn-`**

stall function is used to enable the hardware interrupt and to associate the interrupt service routine with the interrupts from the Device. To identify the source of the interrupt, the interrupt service routine calls and analyzes the response of the **BTICard_EventLogRd** function, just as it did when polling in the previous section. Before returning, the interrupt service routine must call **BTICard_IntClear** to clear the hardware interrupt. See the interrupt example on the software distribution disk.

This page intentionally blank.

APPENDIX A: CSDB FUNCTION REFERENCE

This appendix provides detailed information on the primary CSDB BTIDriver functions for Ballard Technology Devices. The descriptions and examples discussed here are intended for use with programs written in the C language. Users of other languages should contact Ballard for assistance.

Overview of the BTIDriver API

The general naming convention for BTIDriver functions consists of a prefix/category/action format. The functions that make up the BTIDriver library are either specific to a particular avionics databus protocol, or are protocol-independent. The CSDB-specific functions are prefixed by **`BTICSDB_`** (see Table A.1), and the protocol-independent functions are prefixed by **`BTICard_`** (see Table A.2). Functions for other protocols are documented in separate manuals. In this appendix, the function descriptions are listed alphabetically without regard to prefix.

"handle" parameters

All functions in this appendix require a *handle* parameter. The handle is always the last parameter in any function that requires it. The first function called in a program is **`BTICard_CardOpen`**, which returns a card handle (*hCard*). This card handle is passed to **`BTICard_CoreOpen`**, which returns a core handle (*hCore*). Most functions then take this core handle; the only functions that require a card handle are **`BTICard_CoreOpen`** and **`BTICard_CardClose`**. Concepts relating to cards, cores, and handles are described in detail in other protocol manuals.

"ctrlflags" parameters

Many functions have a "ctrlflags" parameter. Each bit controls an option in this bitmask parameter. Constants are defined in the header file for these parameters. The name of a constant reflects the function in which it is used (e.g., CHCFGCSDB_DEFAULT is used in the **`BTICSDB_ChConfig`** function). Option parameters are always first in the parameter list of a function that accepts them. The default options can always be selected by using the ??_DEFAULT constant where ?? depends on the function in which it is used (e.g., CHCFGCSDB_DEFAULT). When multiple options are selected, the constants should be bitwise OR-ed together. The default options are shown in bold in this appendix. Since the default constants are defined as zero, only non-default constants actually need to be included in the OR-ing. The constants defined in the header file should be used by name (not value) in your code since the values are subject to change.

Schedule indices (SCHNDX)

All of the scheduling functions (**`BTICSDB_Sched`**??) return a value of type SCHNDX (Schedule index). These functions append the Command Block index

to the Schedule. This index is a parameter of some of the advanced scheduling functions (e.g., **BTICSDB_SchedCall** and **BTICSDB_SchedBranch**).

"channel" parameters

Some functions take a channel parameter to specify which CSDB channel applies to the function. The header file defines the constants CH0, CH1, etc., which may be used for this purpose. Please refer to the user's manual for your Ballard CSDB Device to determine which channels are receive and which are transmit.

"message" parameters

Message data and related information such as the time-tag are stored in individual Message Records in the Device. All of the message manipulation functions (e.g., **BTICSDB_MsgDataRd**) require a message address parameter that uniquely identifies a Message Record. Several different functions (e.g., **BTICSDB_MsgCreate**, **BTICSDB_FilterSet**, **BTICSDB_FilterDefault**) return the message address.

The message values are arrays of 8-bit values specifying the data of a CSDB message block. The number of valid 8-bit values in the array is specified by the "datacount" parameter of the MSGFIELDSCSDB structure. This parameter may be found using **BTICSDB_MsgBlockRd**.

Table A.1 summarizes the **BTICSDB_** functions and Table A.2 summarizes the **BTICard_** functions. Only the **bolded** functions in the tables below are documented in this appendix. Please consult Ballard's ARINC or MIL-STD-1553 programming manuals for additional information on **BTICard_** functions.

CHANNEL Functions	
Function	**Description**
BTICSDB_ChClear	Clears either a transmit Schedule or a receiver Filter Table
BTICSDB_ChConfig	Configures either a transmit or a receive channel
BTICSDB_ChGetCount	Gets the receiver and transmitter channel count
BTICSDB_ChGetInfo	Gets information about the specified channel on the specified core
BTICSDB_ChIsRcv	Checks if the specified channel is a receiver
BTICSDB_ChIsXmt	Checks if the specified channels is a transmitter
BTICSDB_ChPause	Pauses operation of a channel
BTICSDB_ChPauseCheck	Checks to see if the specified channel is paused
BTICSDB_ChResume	Resumes operation of a previously paused channel
BTICSDB_ChStart	Starts operation of a previously stopped channel
BTICSDB_ChStop	Stops operation of a channel
FILTER Functions	
Function	**Description**
BTICSDB_FilterDefault	Creates default message, and points the entire table to that message
BTICSDB_FilterRd	Reads the message address associated with a filter location
BTICSDB_FilterSet	Creates message, and points specified filter location to that message
BTICSDB_FilterWr	Writes a message address to the specified filter location

Table A.1—CSDB (BTICSDB_) functions (continued on next page)

LIST Functions	
Function	**Description**
BTICSDB_ListDataRd	Reads the next data value associated with a List Buffer
BTICSDB_ListDataWr	Writes the next data value associated with a List Buffer
BTICSDB_ListRcvCreate	Creates a receive List Buffer

BTICSDB_ListStatus	Checks the status of the List Buffer
BTICSDB_ListXmtCreate	Creates a transmit List Buffer
MESSAGE Functions	
Function	**Description**
BTICSDB_MsgBlockRd	Reads an entire Message Record from the Device
BTICSDB_MsgBlockWr	Writes an entire Message Record on the Device
BTICSDB_MsgCreate	Creates and initializes a Message Record
BTICSDB_MsgDataByteRd	Reads the specified data byte associated with a message
BTICSDB_MsgDataByteWr	Writes the specified data byte associated with a message
BTICSDB_MsgDataRd	Reads the data associated with a message
BTICSDB_MsgDataWr	Writes the data associated with a message
BTICSDB_MsgIsAccessed	Returns the value of the Hit bit and then clears it
BTICSDB_MsgValidSet	Sets a non-continuous message valid
PARAMETRIC Functions	
Function	**Description**
BTICSDB_ParamAmplitudeConfig	Configures parametric amplitude control on the specified transmit channel
SCHEDULE Functions	
Function	**Description**
BTICSDB_SchedBranch	Inserts a BRANCH command into the Schedule
BTICSDB_SchedBuild	Builds a Schedule based on message frequency and bus parameters
BTICSDB_SchedCall	Inserts a CALL command into the Schedule
BTICSDB_SchedEntry	Sets the starting point of the Schedule
BTICSDB_SchedGap	Inserts a gap into the Schedule
BTICSDB_SchedHalt	Inserts a HALT Command Block into the Schedule
BTICSDB_SchedLog	Inserts an LOG Command Block into the Schedule
BTICSDB_SchedMsg	Inserts a message into the Schedule
BTICSDB_SchedPause	Inserts a PAUSE Command Block into the Schedule
BTICSDB_SchedRestart	Inserts a RESTART Command Block into the Schedule
BTICSDB_SchedReturn	Inserts a RETURN command into the Schedule

Table A.1—CSDB (BTICSDB_) functions (continued)

CARD Functions	
Function	**Description**
BTICard_CardClose	Disables access to a Device and releases its hardware resources
BTICard_CardOpen	Enables access to a Device and secures hardware resources
BTICard_CardProductStr	Returns the name of the Device
BTICard_CardReset	Resets the Device hardware; destroys all existing configuration data
BTICard_CardResume	Resumes operation of the Device
BTICard_CardStart	Starts operation of the Device
BTICard_CardStop	Stops operation of the Device
BTICard_CardTest	Performs a hardware test on the Device
BTICard_CardTypeStr	Returns the type or model number of the Device
BTICard_CoreOpen	Enables access to a specified core
EVENT Functions	
Function	**Description**
BTICard_EventLogConfig	Enables events and initializes the Event Log List
BTICard_EventLogRd	Reads an entry from the Event Log List
BTICard_EventLogStatus	Checks the status of the Event Log List
I/O Functions	
Function	**Description**
BTICard_ExtDIORd	Reads the value of the specified Digital I/O pin
BTICard_ExtDIOWr	Sets the value of the specified Digital I/O pin
BTICard_ExtLEDRd	Reads the On/Off value of the LED
BTICard_ExtLEDWr	Sets the On/Off value of the LED

Table A.2—Protocol-independent (BTICard_) functions (continued on next page)

INTERRUPT Functions	
Function	**Description**
BTICard_IntClear	Clears the interrupt from the Device (OS-dependent)
BTICard_IntInstall	Associates an event object with interrupts from the Device (OS-dependent)
BTICard_IntUninstall	Removes association between event objects and interrupts (OS-dependent)

IRIG TIME Functions	
Function	**Description**
BTICard_IRIGConfig	Configures the IRIG timer on the specified core
BTICard_IRIGFieldGet??	Returns the ?? field (days, hours, etc.) from an IRIG time-tag
BTICard_IRIGFieldPut??	Writes the ?? field (days, hours, etc.) to an IRIG time-tag
BTICard_IRIGRd	Reads the current value of the IRIG timer on the specified core
BTICard_IRIGSyncStatus	Reports whether the IRIG timer is locked in sync with the IRIG bus
BTICard_IRIGWr	Sets (initializes) the IRIG timer on the specified core
SEQUENTIAL RECORD Functions	
Function	**Description**
BTICard_SeqConfig	Configures the Sequential Monitor
BTICard_SeqFindInit	Initializes the BTICard_SeqFindNext?? functions
BTICard_SeqFindNext	Finds the next message in the Sequential Record
BTICard_SeqFindNext1553	Finds the next MIL-STD-1553 message in the Sequential Record
BTICard_SeqFindNext429	Finds the next ARINC 429 message in the Sequential Record
BTICard_SeqFindNext708	Finds the next ARINC 708 message in the Sequential Record
BTICard_SeqFindNext717	Finds the next ARINC 717 message in the Sequential Record
BTICard_SeqFindNextCSDB	Finds the next CSDB message in the Sequential Record
BTICard_SeqInterval	Sets the interval value if using Interval mode
BTICard_SeqIsRunning	Determines whether the Sequential Record is running
BTICard_SeqLogFrequency	Specifies the period for Sequential Record Event Log List entries
BTICard_SeqRd	Reads data out of the Sequential Record
BTICard_SeqResume	Resumes recording of the Sequential Record where it stopped
BTICard_SeqStart	Starts recording at the beginning of the Sequential Record
BTICard_SeqStatus	Checks the status of the Sequential Record
BTICard_SeqStop	Stops data from being added to the Sequential Record
TIMER Functions	
Function	**Description**
BTICard_TimerClear	Clears the Device timer
BTICard_TimerRd	Reads the current value of the Device timer
BTICard_TimerResolution	Selects a time-tag timer resolution
BTICard_TimerWr	Writes the a value to the Device timer
UTILITY Functions	
Command	**Description**
BTICard_ErrDescStr	Returns the description of the specified error value
BTICard_ValFromAscii	Creates an integer value from an ASCII string
BTICard_ValGetBits	Extracts a bit field from an integer value
BTICard_ValPutBits	Puts a bit field into an integer value
BTICard_ValToAscii	Creates an ASCII string from an integer

Table A.2—Protocol-independent (BTICard_) functions (continued)

The following pages contain descriptions of the BTIDriver functions. The constants in bold in the tables are the default options. Note that the "**BTICard**_" and "**BTICSDB**_" prefixes have been omitted from the headings for easier reading, but all BTIDriver functions in source code must begin with the appropriate prefix

CardClose

```
ERRVAL BTICard_CardClose(
   HCARD hCard                    //Card number of Device
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Disables access to the specified Device and releases the associated hardware resources (e.g., memory and I/O space, interrupt number, and DMA channel). This function does not stop the Device from operating.

WARNINGS

Before a program terminates, this function MUST be called to release the associated hardware resources. This is especially important in Microsoft Windows operating systems.

SEE ALSO

BTICard_CardOpen

CardOpen

```
ERRVAL BTICard_CardOpen(
   LPHCARD lpHandle,          //Pointer to the Device handle
   INT cardnum                //Card number of Device
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

This function enables access to a Device. If BTICard_CardOpen finds the Device that has been assigned *cardnum*, it performs a quick hardware self-test of the Device. Since this function opens the Device handle parameter required by all other functions, this function is always the first BTIDriver function called by a program.

Card numbers are assigned to Devices by the operating system or the user. If only one Device has been installed, the system defaults the card number to zero. How the system assigns card numbers for multiple Devices and how the number can be changed by the user is OS-dependent. See the README.TXT file for your operating system on the distribution disk for more information. A test program for determining the card number(s) is provided on the distribution disk. The card numbers assigned to BTIDriver Devices are specific to BTIDriver-compliant Devices, so there is no conflict when non-BTIDriver-compliant Devices use those same card numbers.

WARNINGS

BTICard_CardClose must be called to release the hardware resources before the program terminates.

SEE ALSO

BTICard_CardClose

CardReset

```
VOID BTICard_CardReset(
   HCARD hCard                  //Device handle
)
```

RETURNS

None.

DESCRIPTION

Stops and performs a hardware reset on the Device. If a message is being processed, the processing is allowed to finish before the Device is halted.

WARNINGS

None.

SEE ALSO

BTICard_CardStart, BTICard_CardStop

CardStart

```
ERRVAL BTICard_CardStart(
   HCARD hCore                    //Device handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Activates all configured channels of the specified Device. The Sequential Monitor and Event Log List are cleared and begin operation at the start of their allocated buffers.

WARNINGS

The Device continues operating even after an application program ends unless BTICard_CardStop halts it. Even after BTICard_CardStart, individual channels may not transmit or receive if they are disabled or paused. See the channel configuration and control functions for each protocol.

SEE ALSO

BTICard_CardStop

CardStop

```
BOOL BTICard_CardStop(
   HCARD hCore                    //Device handle
)
```

RETURNS

TRUE if the Device was active, otherwise FALSE.

DESCRIPTION

Stops operation of the specified Device. If a message is being processed, the processing is allowed to finish before the Device is halted.

WARNINGS

None.

SEE ALSO

BTICard_CardStart

ChClear

```
ERRVAL BTICSDB_ChClear(
   INT channel,                  //Number of the channel
   HCARD hCore                   //Device handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Clears the contents of the specified channel. If the channel is a transmitter, all Command Blocks in the transmit Schedule are deleted. If the channel is a receiver, all filters are deleted. The contents of the Message Records are unaffected. The configuration options previously set by BTICSDB_ChConfig are unchanged.

WARNINGS

None.

SEE ALSO

BTICard_CardReset, BTICSDB_ChConfig

ChConfig

```
ERRVAL BTICSDB_ChConfig(
   ULONG ctrlflags,          //Selects channel options
   USHORT bitrateflag,       //Sets the bit rate
   USHORT bytesperblock,     //Number of bytes per message block
   USHORT framespersecond,   //Number of frames per second
   USHORT blocksperframe,    //Number of blocks per frame
   INT channel,              //Number of the channel
   HCARD hCore               //Device handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Configures the specified transmit or receive channel of the specified Device by performing the following steps:

1. Stops the channel.
2. Clears transmit Schedule for the specified channel (Filter Tables are not affected).
3. Writes Device options defined by *ctrlflags*, *bytesperblock*, *framespersecond*, and *blocksperframe*.
4. Restarts the channel if previously started and not disabled by CHCFGCSDB_INACTIVE.

The *framespersecond* and *blocksperframe* parameters are not used for receive channels.

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
CHCFGCSDB_DEFAULT	Select all default settings (**bold** below)	✓	✓
CHCFGCSDB_PARITYNONE	Select no parity	✓	✓
CHCFGCSDB_PARITYODD	Select odd parity	✓	✓
CHCFGCSDB_PARITYEVEN	Select even parity	✓	✓
CHCFGCSDB_PARITYMARK	Select mark parity	✓	✓
CHCFGCSDB_PARITYSPACE	Select space parity	✓	✓
CHCFGCSDB_ACTIVE	Enable channel activity	✓	✓
CHCFGCSDB_INACTIVE	Disable channel activity	✓	✓
CHCFGCSDB_SEQSEL	Sequential monitoring selected at message level	✓	✓
CHCFGCSDB_SEQALL	Every message will be recorded in the Sequential Record	✓	✓
CHCFGCSDB_NOLOGHALT	No entry will be made in the Event Log List on a HALT command		✓
CHCFGCSDB_LOGHALT	An entry will be made in the Event Log List on a HALT command		✓
CHCFGCSDB_NOLOGPAUSE	No entry will be made in the Event Log List on a PAUSE command		✓
CHCFGCSDB_LOGPAUSE	An entry will be made in the Event Log List on a PAUSE command		✓
CHCFGCSDB_NOLOGERR	No entry will be made in the Event Log List when a decoder detects an error	✓	
CHCFGCSDB_LOGERR	An entry will be made in the Event Log List when a decoder detects an error or goes out of sync	✓	
CHCFGCSDB_TIMETAGOFF	The time-tag option is selected at the message level	✓	✓
CHCFGCSDB_TIMETAG	All messages will record a time-tag.	✓	✓

(Continued from previous page)

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
CHCFGCSDB_ELAPSEOFF	The elapse timing option is selected at the message level	✓	✓
CHCFGCSDB_ELAPSE	All messages will record an elapsed time.	✓	✓
CHCFGCSDB_MAXMINOFF	Max and min repetition rates are selected at the message level	✓	✓
CHCFGCSDB_MAX	All messages will record a max time	✓	✓
CHCFGCSDB_MIN	All messages will record a min time	✓	✓
CHCFGCSDB_MAXMIN	All messages will record a max and a min time	✓	✓
CHCFGCSDB_NOHIT	The Hit Counter is selected at the message level	✓	✓
CHCFGCSDB_HIT	The Hit Counter is selected for all messages (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
CHCFGCSDB_SELFTESTOFF	This channel will transmit/receive on the operational bus and not the self-test bus.	✓	✓
CHCFGCSDB_SELFTEST	This channel will transmit/receive on the internal self-test bus and not the operational bus. Only one transmitter can be on the self-test bus at a time; therefore, only the last transmit channel configured to use the self-test bus will use this option.	✓	✓
CHCFGCSDB_UNPAUSE	The channel will initially be unpaused	✓	✓
CHCFGCSDB_PAUSE	The channel will initially be paused	✓	✓
CHCFGCSDB_NOLOOPMAX	Disable Schedule maximum loop count		✓
CHCFGCSDB_LOOPMAX	Enable Schedule maximum loop count		✓
CHCFGCSDB_BUSINVERTOFF	Disable bus polarity inversion	✓	✓
CHCFGCSDB_BUSINVERT	Enable bus polarity inversion	✓	✓
CHCFGCSDB_MODE422	Enable RS-422 mode (balanced)	✓	✓
CHCFGCSDB_MODE232	Enable RS-232/RS-423 mode (unbalanced)	✓	✓
CHCFGCSDB_SYNCMODE	Synchrounous, scheduled mode (CSDB)	✓	✓
CHCFGCSDB_ASYNCMODE	Asynchrounous mode (general-purpose serial)	✓	✓

The *bitrateflag* sets three channel parameters: a base clock frequency, a prescale value, and a slope setting. The bit rate of the channel can be set by using the provided *bitrateflag* constants in Table A.3.

Alternatively, the user can specify their own bit rate by selecting the base clock frequency using the constants from the Base clock table, the prescale value using the Bit rate equation, and the slope using the constants from the Transmit slope table.

Channels 4 and 12 share a base clock frequency and prescale value. Channels 5 and 13 share a base clock frequency and prescale value. Channels 6, 7, 14, and 15 share a base clock frequency and prescale value. For a custom base frequency contact Ballard Technology (see Section 1.5).

Name	Bit Rate	Prescale	Base Clock	bitrateflag
CSDB	50000	39	20.000M	BITRATECSDB_ HIGHSPEED
	12500	159	20.000M	BITRATECSDB_LOWSPEED
Standard PC Bit Rates	921600	0	9.2160M	BITRATECSDB_921600BPS
	460800	1	9.2160M	BITRATECSDB_460800BPS
	230400	3	9.2160M	BITRATECSDB_230400BPS
	115200	7	9.2160M	BITRATECSDB_115200BPS
	57600	15	9.2160M	BITRATECSDB_57600BPS
	38400	23	9.2160M	BITRATECSDB_38400BPS
	28800	31	9.2160M	BITRATECSDB_28800BPS
	19200	47	9.2160M	BITRATECSDB_19200BPS
	14400	63	9.2160M	BITRATECSDB_14400BPS
	9600	95	9.2160M	BITRATECSDB_9600BPS
	7200	127	9.2160M	BITRATECSDB_7200BPS
	4800	191	9.2160M	BITRATECSDB_4800BPS
	3600	255	9.2160M	BITRATECSDB_3600BPS
	2400	383	9.2160M	BITRATECSDB_2400BPS
	1800	511	9.2160M	BITRATECSDB_1800BPS
	1200	767	9.2160M	BITRATECSDB_1200BPS
	900	1023	9.2160M	BITRATECSDB_900BPS
	600	1535	9.2160M	BITRATECSDB_600BPS
	300	3071	9.2160M	BITRATECSDB_300BPS
Other	400000	4	20.000M	BITRATECSDB_400000BPS
	200000	9	20.000M	BITRATECSDB_200000BPS
	100000	19	20.000M	BITRATECSDB_100000BPS
	50000	39	20.000M	BITRATECSDB_50000BPS
	25000	79	20.000M	BITRATECSDB_25000BPS
	12500	159	20.000M	BITRATECSDB_12500BPS

Table A.3—Software configurable frequencies

(ChConfig continued on next page)

Base Clock	
Constant	**Description**
BITRATECSDB_CLK1	20 MHz
BITRATECSDB_CLK2	9.216 MHz

Base clock table

Transmit Slope	
Constant	**Description**
BITRATECSDB_XMTSLOPEHIGH	High speed rise time
BITRATECSDB_XMTSLOPELOW	Low speed rise time

Transmit slope table

$$\text{Bit Rate} = \frac{CLK_{1\,freq \text{ or } 2\,freq}}{(\text{prescale}+1)\times 10}$$

where $0 \le \text{prescale} \le 4095$, $CLK_{1\,freq} = 20$ MHz, and $CLK_{2\,freq} = 9.216$ MHz

Bit rate equation

WARNINGS

The function clears any previous configuration of the channel.

Each CSDB channel can run at up to its maximum rated speed. However, total throughput is limited by the system configuration (i.e., number of channels, duty cycle, data rate, etc.).

SEE ALSO

BTICard_CardStart, BTICard_CardStop

ChGetCount

```
VOID BTICSDB_ChGetCount(
   LPINT rcvcount,          //Pointer to variable to hold receiver count
   LPINT xmtcount,          //Pointer to variable to hold transmitter count
   HCARD hCore              //Device handle
)
```

RETURNS

None.

DESCRIPTION

Determines the transmitter and receiver channel count, and puts them in the variables pointed to by *rcvcount* and *xmtcount*.

WARNINGS

None.

SEE ALSO

BTICSDB_ChIsRcv, BTICSDB_ChIsXmt

ChGetInfo

```
ULONG BTICSDB_ChGetInfo(
   USHORT infotype,          //Type of information to be returned
   INT channum,              //Number of the channel
   HCARD hCore               //Device handle
)
```

RETURNS

The requested information about the specified channel.

DESCRIPTION

Provides information about the functionality of the specified *channel*. The *infotype* constant listed below may be used to specify the requested information.

infotype		
Constant	**Returns**	**Description**
INFOCSDB_PARAM	1=TRUE 0=FALSE	Channel is parametric. Channel is not parametric.

WARNINGS

None.

SEE ALSO

BTICSDB_ParamAmplitudeConfig

ChIsRcv

```
BOOL BTICSDB_ChIsRcv (
   INT channel,                  //Channel number to test
   HCARD hCore                   //Device handle
)
```

RETURNS

TRUE if the channel is a receiver, or FALSE if it is not a receiver.

DESCRIPTION

Checks to see if the channel number specified by *channel* is a receive channel.

WARNINGS

If this function returns FALSE, do not assume that the channel must then be a transmitter, because the channel may not exist at all. A call to BTICSDB_ChIsXmt must be made to be sure that the channel is a transmitter.

SEE ALSO

BTICSDB_ChIsXmt, BTICSDB_ChGetCount

ChIsXmt

```
BOOL BTICSDB_ChIsXmt (
   INT channel,                  //Channel number to test
   HCARD hCore                   //Device handle
)
```

RETURNS

TRUE if the channel is a transmitter, or FALSE if it is not a transmitter.

DESCRIPTION

Checks to see if the channel number specified by *channel* is a transmit channel.

WARNINGS

If this function returns FALSE, do not assume that the channel must then be a receiver, because the channel may not exist at all. A call to BTICSDB_ChIsRcv must be made to be sure that the channel is a receiver.

SEE ALSO

BTICSDB_ChIsRcv, BTICSDB_ChGetCount

ChPause

```
VOID BTICSDB_ChPause(
   INT channel,                  //Number of the channel
   HCARD hCore                   //Device handle
)
```

RETURNS

None.

DESCRIPTION

Pauses the operation of the channel specified by *channel*. All activity on the channel ceases. The channel remains paused until the channel is resumed by BTICSDB_ChResume. ChConfig initializes the channel as either unpaused (default) or paused.

Note: A transmit channel can also be paused when the Device encounters a PAUSE Command Block in the transmit Schedule.

WARNINGS

Do not confuse this channel pause with either channel enable or the card-level controls. Channel enable is controlled by BTICSDB_ChConfig, BTICSDB_ChStart, and BTICSDB_ChStop. Card-level controls are activated through BTICard_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTICSDB_ChResume, BTICSDB_SchedPause

ChPauseCheck

```
INT BTICSDB_ChPauseCheck(
   INT channel,                  //Number of the channel
   HCARD hCore                   //Device handle
)
```

RETURNS

A non-zero value if the channel is paused, or zero if the channel is not paused.

DESCRIPTION

Determines whether the channel specified by *channel* is paused.

WARNINGS

None.

SEE ALSO

BTICSDB_ChPause, BTICSDB_ChResume

ChResume

```
VOID BTICSDB_ChResume(
   INT channel,                  //Number of the channel
   HCARD hCore                   //Device handle
)
```

RETURNS

None.

DESCRIPTION

Resumes the operation of the channel specified by *channel* after it has been paused by BTICSDB_ChPause or the Device has encountered a PAUSE Command Block in the transmit Schedule. ChConfig initializes the channel as either unpaused (default) or paused. If the Device is running , all activity on the channel will begin. If the Device is stopped, channel activity will begin when the Device is started.

WARNINGS

Do not confuse this channel pause with either channel enable or the card-level controls. Channel enable is controlled by BTICSDB_ChConfig, BTICSDB_ChStart, and BTICSDB_ChStop. Card-level controls are activated through BTICard_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTICSDB_ChPause, BTICSDB_SchedPause

ChStart

```
BOOL BTICSDB_ChStart(
   INT channel,                  //Number of the channel
   HCARD hCore                   //Device handle
)
```

RETURNS

TRUE if the channel was previously enabled, otherwise FALSE.

DESCRIPTION

Enables the operation of the channel specified by *channel*. If it is a transmit channel, the Schedule restarts at the beginning. The channel remains enabled until BTICSDB_ChStop is called or a HALT Command Block is encountered in the transmit Schedule. If the Device is stopped, then channel activity begins when the Device is started with BTICard_CardStart.

BTICSDB_ChStart and BTICSDB_ChStop enable and disable a channel. BTICSDB_ChConfig initializes the channel as either enabled (default) or disabled. These functions allow the channel to be stopped and reconfigured with different settings while other channels are running.

WARNINGS

Do not confuse this channel enable with either channel pause or the card-level controls. Channel pause is controlled by BTICSDB_ChConfig, BTICSDB_ChPause, and BTICSDB_ChResume. Card-level controls are activated through BTICard_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTICSDB_ChStop, BTICard_CardStart, BTICard_CardStop, BTICSDB_SchedHalt

ChStop

```
BOOL BTICSDB_ChStop(
   INT channel,                  //Number of the channel
   HCARD hCore                   //Device handle
)
```

RETURNS

TRUE if the channel was previously enabled, otherwise FALSE.

DESCRIPTION

Disables operation of the channel specified by *channel*. If a message is being sent or received, the processing is allowed to finish before the channel is halted. Use BTICSDB_ChStart to re-enable the channel.

BTICSDB_ChStart and BTICSDB_ChStop enable and disable a channel. BTICSDB_ChConfig initializes the channel as either enabled (default) or disabled. These functions allow the channel to be stopped and reconfigured with different settings while other channels are running.

Note: A transmit channel can also be stopped when the Device encounters a HALT Command Block in the transmit Schedule.

WARNINGS

Do not confuse this channel enable with either channel pause or the card-level controls. Channel pause is controlled by BTICSDB_ChConfig, BTICSDB_ChPause, and BTICSDB_ChResume. Card-level controls are activated through BTICard_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTICSDB_ChStart, BTICSDB_SchedHalt, BTICard_CardStart, BTICard_CardStop

CoreOpen

```
ERRVAL BTICard_CoreOpen(
   LPHCORE lphCore,          //Pointer to a core handle
   INT corenum,              //Core number
   HCARD hCard               //Card handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Enables access to a core (the presence of multiple cores is Device-dependent). BTICard_CardOpen must first be called to obtain the card handle (*hCard*). BTICard_CoreOpen finds the core on the Device specified by *hCard* that has been assigned *corenum*, and returns a handle to that core. BTICard_CoreOpen must be called for each core that you wish to access in your program. BTICard_CardClose will close all cores opened with BTICard_CoreOpen.

If you pass the card handle to a function (such as a channel function) instead of a core handle, it will only access the first (or only) core.

WARNINGS

BTICard_CardOpen must be called before this function. BTICard_CoreOpen must be called for each core that you wish to access in your program.

SEE ALSO

BTICard_CardOpen, BTICard_CardClose

EventLogConfig

```
ERRVAL BTICard_EventLogConfig(
   USHORT ctrlflags          //Selects the configuration options
   USHORT count              //Number of entries in the Event Log List
   HCARD hCore               //Device handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Configures and enables the Event Log List of the Device. The maximum number of entries that may be contained in the Event Log List is set by *count*. *ctrlflags* can be one of the following constants:

ctrlflags	
Constant	**Condition**
LOGCFG_DEFAULT	Selects all default **(bold)** settings
LOGCFG_ENABLE	Enables the Event Log List
LOGCFG_DISABLE	Disables the Event Log List

WARNINGS

None.

SEE ALSO

BTICard_EventLogRd, BTICard_EventLogStatus

EventLogRd

```
ULONG BTICard_EventLogRd(
   LPUSHORT typeval,          //Pointer to variable to receive type value
   LPULONG infoval,           //Pointer to variable to receive info value
   LPINT channel,             //Pointer to variable to receive channel value
   HCARD hCore                //Device handle
)
```

RETURNS

The address of the entry in the Event Log List, or zero if it is empty and there are no entries to read.

DESCRIPTION

Reads the next entry from the Event Log List and advances the pointer. The type of event and channel that generated the entry are passed through *typeval* and *channel*. An information word associated with the event is passed through *infoval*.

The value of *typeval* determines the meaning of the *infoval* value (see table below). Note that the Event Log List records events from all protocols connected to the Device. For other event types consult Ballard's ARINC or MIL-STD-1553 programming manuals.

	typeval	**Description**	***infoval***	**Refer to…**
CSDB	EVENTTYPE_CSDBMSG	Message received or transmitted	Address of the Message Record	BTICSDB_MsgCreate BTICSDB_FilterDefault BTICSDB_FilterSet
	EVENTTYPE_CSDBOPCODE	A transmit Schedule encountered a LOG command	User-assigned tag value	BTICSDB_SchedLog
	EVENTTYPE_CSDBHALT	A transmit Schedule encountered a HALT command	Address of the Schedule entry	BTICSDB_ChConfig BTICSDB_SchedHalt
	EVENTTYPE_CSDBPAUSE	A transmit Schedule encountered a PAUSE command	Address of the Schedule entry	BTICSDB_ChConfig BTICSDB_SchedPause
	EVENTTYPE_CSDBLIST	List Buffer empty or full (underflow or overflow)	List Buffer address	BTICSDB_ListRcvCreate BTICSDB_ListXmtCreate
	EVENTTYPE_CSDBERR	A decoder error was detected	Address of the message that contained the error	BTICSDB_ChConfig
	EVENTTYPE_CSDBSYNCERR	Receive channel lost synchronization	The channel that went out of sync	BTICSDB_ChConfig

WARNINGS

This function should be preceded by a call to BTICard_EventLogConfig. To use this function, it is not necessary to install an interrupt handler.

SEE ALSO

BTICard_EventLogConfig

EventLogStatus

```
INT BTICard_EventLogStatus(
   HCARD hCore                    //Device handle
)
```

RETURNS

The status value of the Event Log List.

DESCRIPTION

Checks the status of the Event Log List without removing an entry. The status value can be tested using the predefined constants below:

Constant	Description
STAT_EMPTY	Event Log List is empty
STAT_PARTIAL	Event Log List is partially filled
STAT_FULL	Event Log List is full
STAT_OFF	Event Log List is off

WARNINGS

When the buffer is full it wraps around and overwrites previous entries.

SEE ALSO

BTICard_EventLogConfig, BTICard_EventLogRd

FilterDefault

```
MSGADDR BTICSDB_FilterDefault(
   ULONG ctrlflags,          //Selects message options
   INT channel,              //Number of receive channel
   HCARD hCore               //Device handle
)
```

RETURNS

Address of the Message Record the function created and placed in the Filter Table.

DESCRIPTION

Creates a Message Record with the options specified in *ctrlflags*, and then sets it as the default Message Record for the channel specified by *channel*. Received messages that do not meet the criteria of specific filters are saved in the default Message Record. If no default filter was created and a message does not match a specific filter, then the message is skipped and not saved in memory.

The options that can be used in *ctrlflags* are listed below. Please note that only the receiver options can be used with this function.

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRTCSDB_DEFAULT	Select all default settings (**bold** below)	✓	✓
MSGCRTCSDB_NOSEQ	This message will not get recorded in the Sequential Record	✓	✓
MSGCRTCSDB_SEQ	This message will get recorded in the Sequential Record	✓	✓
MSGCRTCSDB_NOLOG	This message will not create an entry in the Event Log List	✓	✓
MSGCRTCSDB_LOG	This message will create an entry in the Event Log List	✓	✓
MSGCRTCSDB_NOSKIP	This message will not be skipped	✓	✓
MSGCRTCSDB_SKIP	This message will be skipped, and none of the options will be processed	✓	✓
MSGCRTCSDB_NOTIMETAG	This message will not record a time-tag	✓	✓
MSGCRTCSDB_TIMETAG	This message will record a time-tag	✓	✓
MSGCRTCSDB_NOELAPSE	This message will not record an Elapsed Time	✓	✓
MSGCRTCSDB_ELAPSE	This message will record an Elapsed Time	✓	✓
MSGCRTCSDB_NOMAXMIN	This message will not record maximum and minimum repetition rates	✓	✓
MSGCRTCSDB_MAX	This message will record maximum repetition rates	✓	✓
MSGCRTCSDB_MIN	This message will record minimum repetition rates	✓	✓
MSGCRTCSDB_MAXMIN	This message will record maximum and minimum repetition rates	✓	✓
MSGCRTCSDB_NOHIT	This message will not record a Hit Counter	✓	✓
MSGCRTCSDB_HIT	This message will record a Hit Counter (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
MSGCRTCSDB_WIPE	This data fields of this message will initially be wiped to a value	✓	✓
MSGCRTCSDB_NOWIPE	The data fields of this message will initially be left unchanged	✓	✓
MSGCRTCSDB_WIPE0	The data fields of this message will be wiped with a value of zeros (this option does not get used if MSGCRTCSDB_NOWIPE is used)	✓	✓

(Continued from previous page)

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRTCSDB_WIPESYNC	The data fields of this message will be wiped with the value of 0xA5 (CSDB Sync Character)	✓	✓
MSGCRTCSDB_WIPE1	The data fields of this message will be wiped with a value of ones (this option does not get used if MSGCRTCSDB_NOWIPE is used)	✓	✓

WARNINGS

This function initializes all filters (all Address/SI combinations) for the specified channel. Any filters previously created for the channel are overwritten. Therefore, BTICSDB_FilterDefault MUST precede any calls to BTICSDB_FilterSet.

SEE ALSO

BTICSDB_FilterSet, BTICSDB_FilterRd, BTICSDB_FilterWr

FilterRd

```
MSGADDR BTICSDB_FilterRd(
   INT addrval,              //Address byte value
   INT sival,                //Source Identification value
   INT channel,              //Number of receive channel
   HCARD hCore               //Device handle
)
```

RETURNS

Address of the Message Record the Filter Table is set to for the given parameters.

DESCRIPTION

Reads the address of the Message Record pointed to by the Filter Table for the specified *channel*, *address*, and *sival* combination. The table below shows *sival* equivalents for each source identifier bit pattern. The source identifier is made up of bit 3, bit 1, and bit 0 of the status byte of a message.

SIVAL	
Value	**Status[3:0] (X=don't care)**
0	0X00
1	0X01
2	0X10
3	0X11
4	1X00
5	1X01
6	1X10
7	1X11

WARNINGS

This value reads the address of the Message Record that a filter is pointing to, not the CSDB data. Use BTICSDB_MsgDataRd to read the CSDB data from a Message Record.

SEE ALSO

BTICSDB_FilterWr, BTICSDB_FilterSet, BTICSDB_FilterDefault, BTICSDB_MsgDataRd

FilterSet

```
MSGADDR BTICSDB_FilterSet(
   ULONG ctrlflags,          //Selects message options
   INT addrval,              //Address byte value to receive
   INT simask,               //Source Identification patterns to receive
   INT channel,              //Number of receive channel
   HCARD hCore               //Device handle
)
```

RETURNS

Address of the Message Record the function created and placed in the Filter Table.

DESCRIPTION

Creates a filter for a receive channel by creating a Message Record with the options specified in *ctrlflags*, then setting it as the Message Record for the specified *channel*, *address*, and *simask* values.

Filters are used to sort and save (by address byte and Source Identification bits) messages that are received over a given databus. During operation, when the address and SI bits in a received message match the address and SI information in a specific filter, the message is stored in a specific Message Record location.

The parameter *simask* allows one or more SI combinations to be specified. Using the constants listed below a filter is created for each specified SI. When a combination of SIs are desired, the constants should be OR-ed together.

SIMASK	
Constants	**Description (X=don't care)**
SIx00	Selects only the SI values of X00.
SIx01	Selects only the SI values of X01.
SIx10	Selects only the SI values of X10.
SIx11	Selects only the SI values of X11.
SI000	Selects only the SI value of 000.
SI001	Selects only the SI value of 001.
SI010	Selects only the SI value of 010.
SI011	Selects only the SI value of 011.
SI100	Selects only the SI value of 100.
SI101	Selects only the SI value of 101.
SI110	Selects only the SI value of 110.
SI111	Selects only the SI value of 111.
SIALL	Selects all the SI values.

The options that can be used in ctrlflags are listed below. Please note that only the receiver options can be used with this function.

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRTCSDB_DEFAULT	Select all default settings (**bold** below)	✓	✓
MSGCRTCSDB_NOSEQ	This message will not get recorded in the Sequential Record	✓	✓
MSGCRTCSDB_SEQ	This message will get recorded in the Sequential Record	✓	✓
MSGCRTCSDB_NOLOG	This message will not create an entry in the Event Log List	✓	✓
MSGCRTCSDB_LOG	This message will create an entry in the Event Log List	✓	✓
MSGCRTCSDB_NOSKIP	This message will not be skipped	✓	✓
MSGCRTCSDB_SKIP	This message will be skipped, and none of the options will be processed	✓	✓
MSGCRTCSDB_NOTIMETAG	This message will not record a time-tag	✓	✓
MSGCRTCSDB_TIMETAG	This message will record a time-tag	✓	✓
MSGCRTCSDB_NOELAPSE	This message will not record an Elapsed Time	✓	✓
MSGCRTCSDB_ELAPSE	This message will record an Elapsed Time	✓	✓
MSGCRTCSDB_NOMAXMIN	This message will not record maximum and minimum repetition rates	✓	✓
MSGCRTCSDB_MAX	This message will record maximum repetition rates	✓	✓
MSGCRTCSDB_MIN	This message will record minimum repetition rates	✓	✓
MSGCRTCSDB_MAXMIN	This message will record maximum and minimum repetition rates	✓	✓
MSGCRTCSDB_NOHIT	This message will not record a Hit Counter	✓	✓
MSGCRTCSDB_HIT	This message will record a Hit Counter (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
MSGCRTCSDB_WIPE	This data fields of this message will initially be wiped to a value	✓	✓
MSGCRTCSDB_NOWIPE	The data fields of this message will initially be left unchanged	✓	✓
MSGCRTCSDB_WIPE0	The data fields of this message will be wiped with a value of zeros (this option does not get used if MSGCRTCSDB_NOWIPE is used)	✓	✓
MSGCRTCSDB_WIPESYNC	The data fields of this message will be wiped with the value of 0xA5 (CSDB Sync Character)	✓	✓
MSGCRTCSDB_WIPE1	The data fields of this message will be wiped with a value of ones (this option does not get used if MSGCRTCSDB_NOWIPE is used)	✓	✓

WARNINGS

None.

SEE ALSO

BTICSDB_FilterDefault, BTICSDB_FilterRd, BTICSDB_FilterWr

FilterWr

```
ERRVAL BTICSDB_FilterWr(
   MSGADDR message,          //Message address to write to filter
   INT addrval,              //Address byte value
   INT sival,                //Source Identification value
   INT channel,              //Number of receive channel
   HCARD hCore               //Device handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Writes the message address (*message*) of the Message Record into the Filter Table position specified by the *channel*, *address*, and *sival* combination. The table below shows *sival* equivalents for each source identifier bit pattern. The source identifier is made up of bit 3, bit 1, and bit 0 of the status byte of a message.

SIVAL	
Value	**Status[3:0] (X=don't care)**
0	0X00
1	0X01
2	0X10
3	0X11
4	1X00
5	1X01
6	1X10
7	1X11

This function is most useful to assign multiple addresses to point to one Message Record. After calling BTICSDB_FilterSet for the first address of interest, this function could be called with the message address that was returned by BTICSDB_FilterSet for each of the remaining addresses. This would point all the addresses of interest to one Message Record.

WARNINGS

None.

SEE ALSO

BTICSDB_FilterRd, BTICSDB_FilterSet, BTICSDB_FilterDefault

IntClear

```
VOID BTICard_IntClear(
   HCARD hCore                  //Core handle
)
```

RETURNS

None.

DESCRIPTION

This function clears the interrupt from the core so it is ready for the next interrupt. Typically, the user's worker thread calls this function. Because the core cannot process another interrupt until the current one is cleared, BTICard_IntClear should be called after each interrupt has been processed.

Note: The availability of interrupts is Device-dependent.

WARNINGS

If another interrupt occurs before BTICard_IntClear is called, the new interrupt is lost.

SEE ALSO

BTICard_IntInstall, BTICard_IntUninstall

IntInstall

```
ERRVAL BTICard_IntInstall(
   LPVOID hEvent,              //Handle of a WIN32 event object
   HCARD hCore                 //Core handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

BTICard_IntInstall associates a WIN32 event object with interrupts from the core specified by *hCore*. If the function is successful, any interrupt issued from *hCore* causes the event object specified by *hEvent* to be set to the signaled state.

The user's application must ensure that the event object is set to the unsignaled state before the core issues the first interrupt. This can be done when creating the event object with the WIN32 API function CreateEvent.

Create a worker thread, which immediately goes to sleep by calling a WIN32 API wait function like WaitForSingleObject. When the Device issues an interrupt, the event object is signaled, and the worker thread wakes up to respond to the interrupt. The interrupt is generated whenever an entry is written to the Event Log List.

It is the user's responsibility to clear the interrupt from the core by calling BTICard_IntClear in the worker thread. Note that event objects are never polled.

Note: BTICard_IntInstall should be called separately for each core on the Device, and there should be separate interrupt service threads for each core.

Note: The availability of interrupts is Device-dependent.

WARNINGS

If this function is used, BTICard_IntUninstall MUST be called before the user's program terminates. It removes the association between the Device and the event object.

SEE ALSO

BTICard_EventLogRd, BTICard_IntUninstall

IntUninstall

```
ERRVAL BTICard_IntUninstall(
   HCARD hCore                    //Core handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Removes the association between interrupts from the core specified by *hCore* and WIN32 event objects created by the BTICard_IntInstall function. The Event Log List of the core remains unchanged.

Note: The availability of interrupts is Device-dependent.

WARNINGS

This function must be called before the user's application terminates if BTICard_IntInstall has been called.

SEE ALSO

BTICard_IntInstall

ListDataRd

```
INT BTICSDB_ListDataRd(
   LPBYTE buf,              //Pointer to destination
   INT count,               //Number of bytes to read
   LISTADDR list,           //List from which to read data
   HCARD hCore              //Device handle
)
```

RETURNS

The number of data bytes read from the *list*, or zero if an error occurred or unable to read from the list.

DESCRIPTION

Reads the next data associated with a list. Similar to BTICSDB_MsgDataRd except the data is read from a list. This function copies *count* data bytes to *buf* from the message structure in the list specified by the *list* parameter. The *list* parameter is the value returned when the list was created using BTICSDB_-ListRcvCreate. The position of the message to be read is determined by the mode of the list as follows:

- Circular mode: Not valid for a receive List Buffer.
- FIFO mode: Reads the oldest complete message received.
- Ping-Pong mode: Reads the newest complete message received.

WARNINGS

The list buffer specified by *list* must be configured as a receive List Buffer using BTICSDB_ListRcvCreate.

SEE ALSO

BTICSDB_MsgDataRd, BTICSDB_ListDataWr, BTICSDB_ListRcvCreate, BTICSDB_ListXmtCreate

ListDataWr

```
INT BTICSDB_ListDataWr(
   LPBYTE buf,                //Pointer to data bytes
   INT count,                 //Number of bytes to write
   LISTADDR list,             //List to write new data
   HCARD hCore                //Device handle
)
```

RETURNS

The number of data bytes written to the *list* or zero if an error occurred or unable to write to the list.

DESCRIPTION

Writes the next data associated with a list. Similar to BTICSDB_MsgDataWr except writes to a list. This function copies *count* data bytes from *buf* to the message structure in the list specified by the *list* parameter. The *list* parameter is the value returned when the list was created using BTICSDB_ListXmtCreate. The position to which the message is written in the list is determined by the mode of the list as follows:

- Circular mode: This mode is intended as a preloaded value List Buffer. With a circular List Buffer, this function will write to the next available position and will overwrite data in the buffer when it wraps around.
- FIFO mode: Data is written to one end of the list and is transmitted and removed from the other end of list. This function returns a zero if the list is full.
- Ping-Pong mode: When writing is complete, the data will be used for the next message transmission.

WARNINGS

The list buffer specified by *list* must be configured as a write transmit List Buffer using BTICSDB_ListXmtCreate

SEE ALSO

BTICSDB_MsgDataWr, BTICSDB_ListDataRd, BTICSDB_ListRcvCreate, BTICSDB_ListXmtCreate

ListRcvCreate

```
LISTADDR BTICSDB_ListRcvCreate(
   ULONG ctrlflags,        //Selects list options
   INT count,              //Number of entries in list
   MSGADDR message,        //Message address to associate with List Buffer
   HCARD hCore             //Device handle
)
```

RETURNS

The Device address of the list if successful, otherwise zero.

DESCRIPTION

Creates a receive List Buffer the size of *count* entries. The List Buffer is connected with a Message Record so that the data is processed in the list instead of in the Message Record. The *ctrlflags* specify what type of List Buffer is created and the options associated with the List Buffer. Only FIFO and ping-pong modes are applicable to a receive List Buffer.

ctrlflags	
Constant	**Description**
LISTCRTCSDB_DEFAULT	Select all default settings (**bold** below)
LISTCRTCSDB_FIFO	Selects FIFO mode
LISTCRTCSDB_PINGPONG	Selects ping-pong mode
LISTCRTCSDB_CIRCULAR	Selects circular mode
LISTCRTCSDB_NOLOG	An entry will not be created in the Event Log List when the List Buffer is empty/full
LISTCRTCSDB_LOG	An entry will be created in the Event Log List when the List Buffer is empty/full

WARNINGS

After connecting *message* with a List Buffer, the functions BTICSDB_ListDataRd and BTICSDB_ListDataWr must be used to read or write the data. Do not use BTICSDB_MsgDataRd and BTICSDB_MsgDataWr as they will return incorrect results.

SEE ALSO

BTICSDB_ListXmtCreate, BTICSDB_ListDataWr, BTICSDB_ListDataRd

ListStatus

```
INT BTICSDB_ListStatus(
   LISTADDR list,              //Address of the List Buffer
   HCARD hCore                 //Device handle
)
```

RETURNS

The status value of the specified List Buffer.

DESCRIPTION

Checks the status of the List Buffer specified by *list*. The status value can be tested using the predefined constants below:

Constant	Description
STAT_EMPTY	List Buffer is empty
STAT_PARTIAL	List Buffer is partially filled
STAT_FULL	List Buffer is full
STAT_OFF	List Buffer is off

WARNINGS

None.

SEE ALSO

BTICSDB_ListDataRd, BTICSDB_ListDataWr

ListXmtCreate

```
LISTADDR BTICSDB_ListXmtCreate(
   ULONG ctrlflags,        //Selects list options
   INT count,              //Number of entries in list
   MSGADDR message,        //Message address to associate with List Buffer
   HCARD hCore             //Device handle
)
```

RETURNS

The Device address of the list if successful, otherwise zero.

DESCRIPTION

Creates a transmit List Buffer the size of *count* entries. The List Buffer is connected with a Message Record so that the data is processed in the list instead of in the Message Record. The *ctrlflags* specify what type of List Buffer is created and the options associated with the List Buffer.

ctrlflags	
Constant	**Description**
LISTCRTCSDB_DEFAULT	Select all default settings (**bold** below)
LISTCRTCSDB_FIFO	Selects FIFO mode
LISTCRTCSDB_PINGPONG	Selects ping-pong mode
LISTCRTCSDB_CIRCULAR	Selects circular mode
LISTCRTCSDB_NOLOG	An entry will not be created in the Event Log List when the List Buffer is empty/full
LISTCRTCSDB_LOG	An entry will be created in the Event Log List when the List Buffer is empty/full

WARNINGS

After connecting *message* with a List Buffer, the functions BTICSDB_ListDataRd and BTICSDB_ListDataWr must be used to read or write the data. Do not use BTICSDB_MsgDataRd and BTICSDB_MsgDataWr as they will return incorrect results.

SEE ALSO

BTICSDB_ListRcvCreate, BTICSDB_ListDataWr, BTICSDB_ListDataRd

MsgBlockRd

```
MSGADDR BTICSDB_MsgBlockRd(
   LPMSGFIELDSCSDB msgfields,    //Pointer to destination structure
   MSGADDR message,              //Message from which to read
   HCARD hCore                   //Device handle
)
```

RETURNS

The Message Record address that was read.

DESCRIPTION

Reads an entire Message Record from the Device.

MSGFIELDSCSDB structure		
Field	**Size**	**Description**
msgopt	USHORT	Message options fields. Do not modify these fields.
msgoptext	USHORT	Message optsion fields extended. Do not modify these fields.
burstcount	USHORT	Burst count. Do not modify these fields.
msgact	USHORT	Message activity. See table below for detail.
datacount	USHORT	Number of valid entries in data[] buffer.
listptr	ULONG	List buffer pointer. Used instead of data when in List Buffer mode. Do not modify these fields.
timetag	ULONG	Time-tag value. 32 bits with resolution set by BTICard_TimerResolution.
timetagh	ULONG	Upper 32 bits of the 64-bit IRIG time-tag (if IRIG time is enabled)
hitcount	ULONG	Hit Counter value. Used instead of time-tag when in Hit Counter mode.
maxtime	ULONG	Maximum repetition rate. 32 bits with resolution equal to time-tag resolution.
elapsetime	ULONG	Elapsed time. 32 bits with resolution equal to time-tag resolution. Used instead of maxtime when in Elapsed Time mode.
mintime	ULONG	Minimum repetition rate. 32 bits with resolution equal to time-tag resolution.
userptr	ULONG	Reserved
miscptr	ULONG	Reserved
data[32]	USHORT	Data bytes

The msgact field may be tested by AND-ing the values returned with the constants from the following table:

msgact field	
Constant	**Description**
MSGACTCSDB_CHMASK	The channel number mask value. Shift the result right with MSGACTCSDB_CHSHIFT.
MSGACTCSDB_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACTCSDB_ERR	If set, it signifies that an error occurred in receiving this word. The type of error is defined by the following bits.
MSGACTCSDB_BIT	Bit timing error. An error occurred while decoding the bits of the word.
MSGACTCSDB_NRZ	NRZ error. A non-return to zero error occurred while decoding the edges of the word.
MSGACTCSDB_EDGE	Edge Error. An error occurred while decoding the edges of the word.
MSGACTCSDB_PAR	Parity error. A parity error was detected in the word.
MSGACTCSDB_FRAME	Frame error. The stop bit was incorrect.
MSGACTCSDB_HIT	Signifies that the message has been processed by the firmware (the Hit bit).

WARNINGS

None.

SEE ALSO

BTICSDB_MsgBlockWr, BTICSDB_MsgDataRd, BTICSDB_MsgDataWr, BTICSDB_FilterSet, BTICSDB_FilterDefault

MsgBlockWr

```
MSGADDR BTICSDB_MsgBlockWr(
   LPMSGFIELDSCSDB msgfields,   //Pointer to source structure
   MSGADDR message,             //Message to write to
   HCARD hCore                  //Device handle
)
```

RETURNS

The Message Record address that was written to.

DESCRIPTION

Writes an entire Message Record to the Device. This function is used to modify certain fields in a Message Record after the Message Record was read using BTICSDB_MsgBlockRd. Only the data, hitcount, mintime, and maxtime fields should be modified. All other fields should be restored to the value read. See BTICSDB_MsgBlockRd for a list of the Message Record fields.

WARNINGS

Do not modify any fields other than those listed above.

SEE ALSO

BTICSDB_MsgBlockRd, BTICSDB_MsgDataRd, BTICSDB_MsgDataWr, BTICSDB_MsgCreate, BTICSDB_FilterSet, BTICSDB_FilterDefault

MsgCreate

```
MSGADDR BTICSDB_MsgCreate(
   ULONG ctrlflags,          //Selects message options
   HCARD hCore               //Device handle
)
```

RETURNS

The Device address of the Message Record if successful, otherwise zero.

DESCRIPTION

Allocates memory for a Message Record and initializes the record with the options specified in *ctrlflags*. The options that can be used in *ctrlflags* are listed below. Please note that only the transmitter options can be used with this function.

Ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRTCSDB_DEFAULT	Select all default settings (**bold** below)	✓	✓
MSGCRTCSDB_NOSEQ	This message will not get recorded in the Sequential Record	✓	✓
MSGCRTCSDB_SEQ	This message will get recorded in the Sequential Record	✓	✓
MSGCRTCSDB_NOLOG	This message will not create an entry in the Event Log List	✓	✓
MSGCRTCSDB_LOG	This message will create an entry in the Event Log List	✓	✓
MSGCRTCSDB_NOSKIP	This message will not be skipped	✓	✓
MSGCRTCSDB_SKIP	This message will be skipped, and none of the options will be processed	✓	✓
MSGCRTCSDB_NOTIMETAG	This message will not record a time-tag	✓	✓
MSGCRTCSDB_TIMETAG	This message will record a time-tag	✓	✓
MSGCRTCSDB_NOELAPSE	This message will not record an Elapsed Time	✓	✓
MSGCRTCSDB_ELAPSE	This message will record an Elapsed Time	✓	✓
MSGCRTCSDB_NOMAXMIN	This message will not record maximum and minimum repetition rates	✓	✓
MSGCRTCSDB_MAX	This message will record maximum repetition rates	✓	✓
MSGCRTCSDB_MIN	This message will record minimum repetition rates	✓	✓
MSGCRTCSDB_MAXMIN	This message will record maximum and minimum repetition rates	✓	✓
MSGCRTCSDB_NOHIT	This message will not record a Hit Counter	✓	✓
MSGCRTCSDB_HIT	This message will record a Hit Counter (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
MSGCRTCSDB_WIPE	This data fields of this message will initially be wiped to a value	✓	✓
MSGCRTCSDB_NOWIPE	The data fields of this message will initially be left unchanged	✓	✓
MSGCRTCSDB_WIPE0	The data fields of this message will be wiped with a value of zeros (this option does not get used if MSGCRTCSDB_NOWIPE is used)	✓	✓
MSGCRTCSDB_WIPESYNC	The data fields of this message will be wiped with the value of 0xA5 (CSDB Sync Character)	✓	✓
MSGCRTCSDB_WIPE1	The data fields of this message will be wiped with a value of ones (this option does not get used if MSGCRTCSDB_NOWIPE is used)	✓	✓

(Continued on next page)

(Continued from previous page)

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRTCSDB_CONT	Selects continuous mode.		✓
MSGCRTCSDB_NONCONT	Selects non-continuous mode.		✓
MSGCRTCSDB_BURST	Selects burst mode.		✓

WARNINGS

None.

SEE ALSO

BTICSDB_MsgDataRd, BTICSDB_MsgDataWr, BTICSDB_MsgBlockRd, BTICSDB_MsgBlockWr

MsgDataByteRd

```
BYTE BTICSDB_MsgDataByteRd(
   INT index,                  //Byte offset, zero based
   MSGADDR msgaddr,            //Message from which to read
   HCARD hCore                 //Device handle
)
```

RETURNS

Data byte at *index* offset into the message record data buffer array, otherwise zero.

DESCRIPTION

Reads the byte at *index* offset into the message record data buffer array specified by *msgaddr*.

WARNINGS

The data buffer array of any message record has a maximum count of 32. However, the "datacount" field from BTICSDB_MsgBlockRd, or the *bytesperblock* parameter from BTICSDB_ChConfig, should be used as the maximum count of relevant data in a given message record.

SEE ALSO

BTICSDB_MsgDataWr, BTICSDB_MsgCreate, BTICSDB_MsgBlockRd, BTICSDB_MsgBlockWr, BTICSDB_FilterSet, BTICSDB_FilterDefault

MsgDataByteWr

```
INT BTICSDB_MsgDataByteWr(
   BYTE data,                  //Data value to write
   INT index,                  //Offset to write, zero based
   MSGADDR msgaddr,            //Message to receive new data
   HCARD hCore                 //Device handle
)
```

RETURNS

Zero if an error occurs, or one if successful.

DESCRIPTION

Writes the data byte associated with a message to the given *index* offset. This function copies data from *data* to the message structure specified by *msgaddr*.

WARNINGS

The data buffer array of any message record has a maximum count of 32. However, the "datacount" field from BTICSDB_MsgBlockRd, or the *bytesperblock* parameter from BTICSDB_ChConfig, should be used as the maximum count of relevant data in a given message record.

SEE ALSO

BTICSDB_MsgDataRd, BTICSDB_MsgCreate, BTICSDB_MsgBlockRd, BTICSDB_MsgBlockWr

MsgDataRd

```
INT BTICSDB_MsgDataRd(
   LPBYTE buf,                //Pointer to data bytes
   INT count,                 //Number of words to read
   MSGADDR message,           //Message from which to read
   HCARD hCore                //Device handle
)
```

RETURNS

Number of bytes successfully read from the message structure, otherwise zero.

DESCRIPTION

Reads the data associated with a message. This function copies up to *count* data bytes to *buf* from the message structure specified by *message*.

WARNINGS

None.

SEE ALSO

BTICSDB_MsgDataWr, BTICSDB_MsgCreate, BTICSDB_MsgBlockRd, BTICSDB_MsgBlockWr, BTICSDB_FilterSet, BTICSDB_FilterDefault

MsgDataWr

```
INT BTICSDB_MsgDataWr(
   LPBYTE buf,                //Pointer to data bytes
   INT count,                 //Number of bytes to write
   MSGADDR message,           //Message to receive new data
   HCARD hCore                //Device handle
)
```

RETURNS

Number of bytes successfully written to the message structure, otherwise zero.

DESCRIPTION

Writes the data associated with a message. This function copies up to *count* data bytes from *buf* to the message structure specified by *message*.

WARNINGS

None.

SEE ALSO

BTICSDB_MsgDataRd, BTICSDB_MsgCreate, BTICSDB_MsgBlockRd, BTICSDB_MsgBlockWr

MsgIsAccessed

```
BOOL BTICSDB_MsgIsAccessed(
   MSGADDR message,          //Message to receive new data
   HCARD hCore               //Device handle
)
```

RETURNS

TRUE if a message has been accessed (if the Hit bit is set), otherwise FALSE.

DESCRIPTION

This function indicates that a Message Record has been processed by either transmission or reception of a message. BTICSDB_MsgIsAccessed returns the value of the Hit bit, then clears it.

WARNINGS

None.

SEE ALSO

BTICSDB_MsgDataRd, BTICSDB_MsgCreate, BTICSDB_MsgBlockRd, BTICSDB_MsgBlockWr, BTICSDB_FilterSet, BTICSDB_FilterDefault

MsgValidSet

```
USHORT BTICSDB_MsgValidSet(
   MSGADDR message,          //Message to set as valid
   HCARD hCore               //Device handle
)
```

RETURNS

The previous value of the valid bit.

DESCRIPTION

This function sets the valid bit of a given message specified by *message*. Continuous messages are always valid by default while non-continuous messages must be explicitly set valid by the user. In general a non-continuous message is created using BTICSDB_MsgCreate. The data is written using BTICSDB_MsgDataWr (or left as initialized by BTICSDB_MsgCreate) and must be set valid by BTICSDB_MsgValidSet. Once a non-continuous message is encountered in the transmit Schedule and is transmit on the bus, its valid bit is cleared (Burst messages are transmit in 16 consecutive frames before their valid bits are cleared).

WARNINGS

None.

SEE ALSO

BTICSDB_MsgDataRd, BTICSDB_MsgDataWr, BTICSDB_MsgCreate

ParamAmplitudeConfig

```
ERRVAL BTICSDB_ParamAmplitudeConfig(
   ULONG configval,          //Configuration options to set
   USHORT dacval             //12-bit digital-analog converter value
   INT xmtchan               //Transmit channel number
   HCORE hCore               //Core handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Enables parametric amplitude control as defined by *configval* (see table below) on the transmit channel specified by *xmtchan* and sets the digital-to-analog converter to *dacval*. If this parametric control is not used or is disabled, then the amplitude reverts to default (full) amplitude. Variable transmit amplitude capability is Device-dependent.

configval	
Constant	**Description**
PARAMCFGCSDB_DEFAULT	Select all default settings (**bold** below)
PARAMCFGCSDB_AMPLON	Enables parametric amplitude control
PARAMCFGCSDB_AMPLOFF	Disables parametric amplitude control

WARNINGS

This function may only be used with a CSDB transmit channel with parametric capability.

SEE ALSO

BTICSDB_ParamBitRateConfig, BTICSDB_ChGetInfo

SchedBranch

```
SCHNDX BTICSDB_SchedBranch(
   USHORT condition,          //Condition for branch
   SCHNDX destindex,          //Destination index for branch
   INT channel,               //Channel number of transmitter
   HCARD hCore                //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a conditional BRANCH Command Block to the current end of the Schedule. A conditional BRANCH Command Block causes the Device to branch to the index in the Schedule specified by *destindex* if *condition* evaluates as TRUE.

The *condition* flags listed below may be used to specify the branch condition. The existence of the digital I/O signals (DIO) is Device-dependent.

condition	
Constant	**Description**
CONDCSDB_ALWAYS	Always branch
CONDCSDB_DIO1ACT	Branch if DIO1 is active
CONDCSDB_DIO1NACT	Branch if DIO1 is inactive
CONDCSDB_DIO2ACT	Branch if DIO2 is active
CONDCSDB_DIO2NACT	Branch if DIO2 is inactive
CONDCSDB_DIO3ACT	Branch if DIO3 is active
CONDCSDB_DIO3NACT	Branch if DIO3 is inactive
CONDCSDB_DIO4ACT	Branch if DIO4 is active
CONDCSDB_DIO4NACT	Branch if DIO4 is inactive

WARNINGS

A call to BTICSDB_ChConfig must precede this function. When creating subroutines, BTICSDB_SchedEntry needs to be called to point to the main section. The Command Block pointed to by *destindex* must have been previously inserted in the Schedule.

SEE ALSO

BTICSDB_SchedEntry, BTICSDB_SchedCall, BTICSDB_SchedReturn

SchedBuild

```
ERRVAL BTICSDB_SchedBuild(
   INT nummsgs,              //Number of messages to Schedule
   LPMSGADDR msgs[],         //Array of message addresses
   LPINT freq[],             //Array of message frequencies
   INT interblockgap,        //Gap time to be inserted between blocks
   INT rsvdblocks,           //Number of unassigned blocks
   INT channel,              //Channel number of transmitter
   HCARD hCore               //Device handle
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Clears any transmit Schedule for the specified channel and creates a new transmit Schedule that sequences message blocks within frames. The new Schedule will consist of *nummsgs* messages, each transmitted at an interval specified by the *freq* interval array (in units of frames). Each frame will preserve *rsvdblocks* number of blocks unassigned. *msgs* points to an array of message addresses, each previously generated by a call to BTICSDB_MsgCreate.

The nth element of the *msgs* array uses the nth element of the *freq* intervals array to create the Schedule.

The function Schedules messages and gaps to generate the specified transmit intervals. If the Schedule cannot be generated, an error is returned.

WARNINGS

A call to BTICSDB_ChConfig must precede this function as well as a call to BTICSDB_MsgCreate for each message to be Scheduled.

SEE ALSO

BTICSDB_SchedMsg, BTICSDB_SchedMsgEx, BTICSDB_SchedGap

SchedCall

```
SCHNDX BTICSDB_SchedCall(
   USHORT condition,          //Condition for call
   SCHNDX destindex,          //Destination index of subroutine
   INT channel,               //Channel number of transmitter
   HCARD hCore                //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a conditional CALL Command Block to the current end of the Schedule. A conditional CALL Command Block causes the Schedule to execute the subroutine at the index in the Schedule specified by *destindex* if *condition* evaluates as TRUE.

The *condition* flags listed below may be used to specify the call condition. The existence of the digital I/O signals (DIO) is Device-dependent.

condition	
Constant	**Description**
CONDCSDB_ALWAYS	Always call
CONDCSDB_DIO1ACT	Call if DIO1 is active
CONDCSDB_DIO1NACT	Call if DIO1 is inactive
CONDCSDB_DIO2ACT	Call if DIO2 is active
CONDCSDB_DIO2NACT	Call if DIO2 is inactive
CONDCSDB_DIO3ACT	Call if DIO3 is active
CONDCSDB_DIO3NACT	Call if DIO3 is inactive
CONDCSDB_DIO4ACT	Call if DIO4 is active
CONDCSDB_DIO4NACT	Call if DIO4 is inactive

WARNINGS

A call to BTICSDB_ChConfig must precede this function. After creating subroutines, BTICSDB_SchedEntry needs to be called to point to the main section. Every use of BTICSDB_SchedCall must have a corresponding BTICSDB_SchedReturn. The Command Block pointed to by *destindex* must have been previously inserted in the Schedule.

SEE ALSO

BTICSDB_SchedEntry, BTICSDB_SchedBranch, BTICSDB_SchedReturn

SchedEntry

```
SCHNDX BTICSDB_SchedEntry(
   INT channel,              //Channel number of transmitter
   HCARD hCore               //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Sets the next available location in the Schedule as the beginning of the Schedule. This operation is only necessary if subroutines are used in a Schedule.

To create a Schedule with subroutines, first define the subroutines by calling the desired Schedule functions while saving the returned Schedule indices. Then call BTICSDB_SchedEntry to set the starting point of the Schedule. Then build the main part of the Schedule by calling the other Schedule functions that include the commands that call the subroutines (i.e., BTICSDB_SchedCall and BTICSDB_SchedBranch).

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICSDB_SchedBranch, BTICSDB_SchedCall, BTICSDB_SchedReturn

SchedGap

```
SCHNDX BTICSDB_SchedGap(
   USHORT gap,                  //Gap value in bit times
   INT channel,                 //Channel number of transmitter
   HCARD hCore                  //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a GAP Command Block to the current end of the Schedule. When a GAP Command Block is encountered in the Schedule, it triggers the transmission of *gap* number of bit times of idle bus.

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICSDB_SchedMsg, BTICSDB_SchedGapFixed, BTICSDB_SchedGapList

SchedHalt

```
SCHNDX BTICSDB_SchedHalt(
   INT channel,              //Channel number of transmitter
   HCARD hCore               //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a HALT Command Block to the current end of the Schedule. A HALT Command Block stops the Schedule until the channel is re-enabled using BTICSDB_ChStart. When a HALT Command Block is encountered, it has the same effect as executing the BTICSDB_ChStop function.

Note: Execution of this function does NOT halt the Schedule. The Schedule is halted only when the resulting Schedule executes the HALT Command Block after BTICard_CardStart starts the Device.

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICSDB_SchedPause, BTICSDB_ChStart, BTICSDB_ChStop

SchedLog

```
SCHNDX BTICSDB_SchedLog(
   USHORT condition,          //Value to test
   USHORT tagval,             //Event tag value
   INT channel,               //Channel number of transmitter
   HCARD hCore                //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a conditional LOG Command Block to the current end of the Schedule. A conditional LOG Command Block causes the Device to generate an Event Log List entry if *condition* evaluates as TRUE. The event type placed in the Event Log List is EVENTTYPECSDB_OPCODE and the user-specified value *tagval* is used as the info value. Entries are read out of the Event Log List using BTICard_EventLogRd.

The *condition* flags listed below may be used to specify the Event condition. The existence of the digital I/O signals (DIO) is Device-dependent.

condition	
Constant	**Description**
CONDCSDB_ALWAYS	Always log an entry in the Event Log List
CONDCSDB_DIO1ACT	Log if DIO1 is active
CONDCSDB_DIO1NACT	Log if DIO1 is inactive
CONDCSDB_DIO2ACT	Log if DIO2 is active
CONDCSDB_DIO2NACT	Log if DIO2 is inactive
CONDCSDB_DIO3ACT	Log if DIO3 is active
CONDCSDB_DIO3NACT	Log if DIO3 is inactive
CONDCSDB_DIO4ACT	Log if DIO4 is active
CONDCSDB_DIO4NACT	Log if DIO4 is inactive

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICard_EventLogRd

SchedMsg

```
SCHNDX BTICSDB_SchedMsg(
   MSGADDR message,          //Address of message
   INT channel,              //Channel number of transmitter
   HCARD hCore               //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a MESSAGE command Block to the current end of the Schedule. When a MESSAGE command Block is encountered in the Schedule, the message data specified by *message* are loaded into the transmit channel's encoder and the message is transmitted.

Note: Execution of this function does NOT transmit the message. The message is transmitted only when the resulting Schedule is executed after the channel is enabled and the Device is started.

WARNINGS

A call to BTICSDB_ChConfig must precede this function. In addition, the message must have been created with BTICSDB_MsgCreate.

SEE ALSO

BTICSDB_SchedMsgEx, BTICSDB_MsgCreate

SchedPause

```
SCHNDX BTICSDB_SchedPause(
   INT channel,                  //Channel number of transmitter
   HCARD hCore                   //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a PAUSE Command Block to the current end of the Schedule. A PAUSE Command Block pauses the channel specified by *channel* until the Device is unpaused using BTICSDB_ChResume. When a PAUSE Command Block is encountered, it has the same effect as executing the BTICSDB_ChPause function.

Note: Execution of this function does NOT pause the channel. The channel is paused only when the resulting Schedule is executed after the channel is enabled and the Device is started.

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICSDB_ChPause, BTICSDB_ChResume, BTICSDB_SchedHalt, BTICSDB_SchedRestart

SchedRestart

```
SCHNDX BTICSDB_SchedRestart(
   INT channel,              //Channel number of transmitter
   HCARD hCore               //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a RESTART Command Block to the current end of the Schedule. A RESTART Command Block restarts the Schedule back at the beginning. A RESTART Command Block is automatically appended to the end of the Schedule, so this function does not need to be called for simple Schedules.

Note: Execution of this function does NOT restart the Schedule. The Schedule is restarted only when the resulting Schedule is executed after the channel is enabled and the Device is started.

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICSDB_SchedEntry

SchedReturn

```
SCHNDX BTICSDB_SchedReturn(
   INT channel                  //Channel number of transmitter
   HCARD hCore                  //Device handle
)
```

RETURNS

The index of the appended Command Block if successful, or a negative value if an error occurs.

DESCRIPTION

Appends a RETURN Command Block to the current end of the Schedule. A RETURN Command Block returns the Schedule to the point at which the last CALL command was made. For every CALL command there must be a RETURN command to insure proper operation.

WARNINGS

A call to BTICSDB_ChConfig must precede this function.

SEE ALSO

BTICSDB_SchedCall

SeqFindNext

```
ERRVAL BTICard_SeqFindNext(
   LPUSHORT *pRecord,          //Address of pointer
   LPUSHORT seqtype,           //Pointer to variable to receive type value
   LPSEQFINDINFO sfinfo        //Pointer to structure
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

This function finds the next record (**pRecord*) in the Sequential Record buffer (regardless of protocol). The protocol for that record is indicated by *seqtype* as shown below. The *sfinfo* structure is also updated.

seqtype	Description
SEQTYPE_429	Sequential Record type is ARINC 429.
SEQTYPE_717	Sequential Record type is ARINC 717.
SEQTYPE_1553	Sequential Record type is MIL-STD-1553.
SEQTYPE_708	Sequential Record type is ARINC 708.
SEQTYPE_CSDB	Sequential Record type is CSDB.
SEQTYPE_USER	Sequential Record type is user-defined.

Calling this function repeatedly steps through the records one at a time until the end of the buffer is reached (at which time it returns an error value). Thus, messages can be individually saved to disk, displayed on screen, etc. To handle the record data, cast the **pRecord* value to a structure pointer defined in the protocol-specific BTICard_SeqFindNext?? functions.

WARNINGS

To make this function start its search at the beginning of the Sequential Record buffer, the *sfinfo* structure must first be initialized with BTICard_SeqFindInit. Otherwise, it finds the next record from where it left off.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqFindNextCSDB

```
ERRVAL BTICard_SeqFindNextCSDB(
   LPSEQRECORDCSDB *pRecord,     //Address of pointer to a structure
   LPSEQFINDINFO sfinfo          //Pointer to structure
)
```

RETURNS

A negative value if an error occurs, or zero if successful.

DESCRIPTION

Finds the next CSDB record in the Sequential Record buffer and updates **pRecord* to point to that record. This function uses and updates data from the *sfinfo* structure. Calling this function repeatedly returns the CSDB records one at a time until the end of the buffer is reached (at which time it returns an error value). Thus, messages can be individually saved to disk, displayed on screen, etc. Using the predefined Sequential Record structure SEQRECORDCSDB allows for easy handling of the data.

SEQRECORDCSDB structure		
Field	**Size**	**Description**
type	USHORT	The protocol and version number of this record
timestamp	ULONG	The time-tag value
timestamph	ULONG	The time-tag value
activity	USHORT	Activity (see table below for details)
datacount	USHORT	Number of data bytes
data	USHORT	Array of data bytes (don't exceed data[datacount-1])

The activity field may be tested by AND-ing the value returned with the constants from the table below:

CSDB activity field	
Constant	**Description**
MSGACTCSDB_CHMASK	The channel number mask value. Shift the result right with MSGACTCSDB_CHSHIFT.
MSGACTCSDB_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACTCSDB_ERR	If set, it signifies that an error occurred in receiving this word. The type of error is defined by the following bits.
MSGACTCSDB_BIT	Bit timing error. An error occurred while decoding the bits of the word.
MSGACTCSDB_NRZ	NRZ error. A non-return to zero error occurred while decoding the edges of the word.
MSGACTCSDB_EDGE	Edge Error. An error occurred while decoding the edges of the word.
MSGACTCSDB_PAR	Parity error. A parity error was detected in the word.
MSGACTCSDB_FRAME	Frame error. The stop bit was incorrect.
MSGACTCSDB_HIT	Signifies that the message has been processed by the firmware (the Hit bit).

Extract the channel number from the activity word by AND-ing the activity field with MSGACTCSDB_CHMASK and right-shifting the result by MSGACTCSDB_CHSHIFT. The resulting value is the channel number associated with the CSDB record.

```
channel = (activity & MSGACTCSDB_CHMASK) >>
MSGACTCSDB_CHSHIFT;
```

WARNINGS

Must be preceded by a call to BTICard_SeqFindInit.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit

APPENDIX B: MULTI-PROTOCOL / DEVICE PROGRAMS

A single software application can be written to simultaneously operate many similar or dissimilar Ballard BTIDriver-compliant products, each supporting a single or multiple avionics databus protocols. This appendix provides information needed to write software programs to control multiple Devices and Devices that support more than one protocol.

Programming Rules

Guidelines for writing multi-Device and multi-protocol programs are summarized in the following rules. The discussion in the rest of this appendix further explains these rules.

1. A card number for each Device is assigned by the operating system. If only one BTIDriver-compliant Device exists on the system, it is assigned card number zero (0) by the operating system.
2. A core number for each core on the Device is set by the architecture of the Device. If only one core exists on the Device, it is core number zero.
3. A test utility is provided with the Device for indicating and associating the card number with each individual Device and the core number of each core. A utility for reassigning the card number may also be included with the Device. The card numbers assigned to BTIDriver-compliant Devices are specific to them, so there is no conflict when devices that are not BTIDriver-compliant use those same card numbers.
4. The card handle returned by BTICard_CardOpen is passed to BTICard_CoreOpen to obtain the core handle used by all channels and all protocols on that core.
5. The recommended programming practice is to use the card handle only in BTICard_CoreOpen and BTICard_CardClose (i.e., to obtain core handles and to release the resources back to the operating system at the end of the program). All other functions needing a handle should use the core handle.
6. If a card handle is used in place of a core handle, it has the same effect as when the handle for core number zero is used. Programs for single-core Devices can be written without using core handles, but they would be more easily ported to other Devices by following the recommendation of using core handles.
7. Card functions (those prefixed with BTICard_) are shared with all protocols and channels on the core specified by the core handle. For instance, BTICard_CardStart starts all channels on the core (independent of protocol). Note that using a card handle with this function only starts channels on core number zero.
8. Different protocol functions may be interleaved in the program between the common BTICard_ functions.

BTICard_ Functions

BTICard_ functions are common to all protocols supported by the Device. When a BTICard_ function is used, all protocols on the Device specified by the handle are affected. Programs supporting different protocols may be combined into a single program by interleaving the protocol-specific functions with common BTICard_ functions. A normal application would use BTICard_CardOpen and BTICard_CardClose only once for each Device. Similarly, Card functions like BTICard_CardStart and BTICard_CardStop apply to all channels and protocols on the Device.

Sequential Record

Each Device has one Sequential Record, independent of how many different protocols it supports. The format of individual records within the Sequential Record differs between protocols. There are two ways of scanning through a Sequential Record: by protocol-specific records or by every record. To scan by protocol, use the BTICard_SeqFindNext?? function to find the next record with the ?? protocol. For instance, the BTICard_SeqFindNextCSDB and BTICard_SeqFindNext429 functions are used to find the next CSDB or ARINC 429 record respectively. To scan through every record, use the BTICard_SeqFindNext function, which finds the next record and returns the type (CSDB, 429, etc.). The different BTICard_SeqFindNext?? functions should not be mixed within a sequence without first using BTICard_SeqFindInit. Note that the BTICard_SeqFindNext?? functions do not use a handle, so they do not access the Device. They work from a copy of the Sequential Record in the computer's memory. Thus, they may be used to processes a Sequential Record that had been saved to a hard disk.

Event Log List

As with the Sequential Record, there is one Event Log List per Device, independent of how many protocols are supported. However, all records in the Event Log List have the same format. To determine the cause of the event and the protocol associated with it, test the type value passed through BTICard_EventLogRd. There are some event types that are common between protocols and some that are unique to specific protocols.

Using Multiple Devices

A program that uses more than one Device can be viewed as a combination of programs for the individual Devices. Every BTIDriver function in the individual programs would appear in the combined program and may be interleaved so as to provide the desired functionality. All BTICard_ functions affect only the Device specified by the handle (e.g., each Device needs it own BTICard_CardOpen, BTICard_CardStart, BTICard_CardStop, and BTICard_CardClose functions). If interrupts are used, there should be separate interrupt service threads for each Device. In a similar way, non-BTIDriver-compliant devices may be combined into the program.

APPENDIX C: SPECIFICATIONS

Modules:

- Module 433: 4R/4T CSDB
- Module 437: 4R/4T CSDB with 4R/4T ARINC 429

General:

- Each CSDB module has 4 receive and 4 transmit (4R/4T) channels
- Programmable parity for each CSDB channel (odd/even/mark/space/none)
- Programmable speed options:
  - LO: 12.5 Kbps
  - HI: 50 Kbps
  - Standard PC bit rates
  - Other
- Buffering schemes facilitate data handling
  - Single buffer is the default (receive/transmit)
  - Ping-pong double buffers ensure data integrity (receive/transmit)
  - Circular lists transmit a repeated pattern, such as a sine wave (transmit)
  - FIFO list buffers can handle sequences of data (receive/transmit)
- Internal wrap-around self-test bus facilitates built-in test and diagnostics

Transmitters:

- Vary the transmit amplitude under software control
- Messages scheduled according to bus parameters
- Continuous, non-continuous, and burst transmission supported
- Messages can be tagged for logging/interrupts

Receivers:

- Automatically synchronizes with frame sync block
- Programmable filtering by address byte and source identifier field (status byte)
- Automatic error detection:
  - Bit error
  - Parity error
  - Framing error
- Detected errors can be logged or generate interrupts

Sequential Monitor:

- Create a sequential record in on-board memory or stream to file (with a simple program)
- Monitor concurrently with transmitter/receiver operation
- Monitor CSDB and ARINC 429 concurrently
- Monitor recording modes:
  - Circular
  - Fill and halt
- Each entry includes the address byte, status, data, channel number, and time-tag

Time-Tags:

- Use 32-bit board timer or 64-bit IRIG timer (displays day/hour/min/sec/ms/µs)
- IRIG timer options:
  - Select IRIG-B or IRIG-A format
  - Generate IRIG signal or synchronize to an IRIG signal (on-board or external)
  - Initialize timer to time of day or other value

Interrupts/Logging:

- Configurable event log can be polled and can generate interrupts to the host PC
- The following events may be user-selected for logging/interrupts:
  - When the monitor is full or halts
  - On a user-specified frequency of monitored messages
  - When tagged messages are sent or received
  - When a message error is detected
  - When a list buffer is empty or full

APPENDIX D: REVISION HISTORY

The following revisions have been made to this manual:

Rev A. Date: July 15, 2005

Original release of this manual.

This page intentionally blank.