ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Astronics / Ballard Technology LP429-5/8R8T* at our website: *Click HERE*

ARINC 429
PROGRAMMING MANUAL

for BTIDriver-Compliant Devices

January 6, 2010
Rev. C

Copyright © 2001–2010
by

Ballard TECHNOLOGY

11400 Airport Road
Everett, WA 98204 USA

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

MA132-20100106

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

COPYRIGHT NOTICE

Copyright © 2001–2010 by Ballard Technology, Inc. Ballard Technology's permission to copy and distribute this manual is for the purchaser's private use only and is conditioned upon purchaser's use and application with the Ballard Technology hardware and/or software that was shipped with this manual. No commercial resale or outside distribution rights are allowed by this notice. This material remains the property of Ballard Technology, Inc. All other rights reserved by Ballard Technology, Inc.

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

Windows® is a registered trademark of Microsoft Corporation. OmniBus® and CoPilot® are registered trademarks of Ballard Technology, Inc. BTIDriver™ is a trademark of Ballard Technology, Inc. All other product names or trademarks are property of their respective owners.



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

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

TABLE OF CONTENTS

1. INTRODUCTION **1-1**

1.1 ARINC 429 Overview 1-1
1.2 BTIDriver for ARINC 429 1-1
1.3 CoPilot 1-2
1.4 How to Use This Manual 1-3
1.5 Technical Support and Customer Service 1-3
1.6 Updates 1-3

2. PROGRAMMING BASICS **2-1**

2.1 Terminology 2-1
2.2 Getting Started 2-2
2.3 Steps a Program Must Perform 2-2
2.4 Handles, Cards, and Cores 2-3
2.5 Receiver Example 2-5
2.6 Transmitter Example 2-6
2.7 Monitor Example 2-8

3. ADVANCED OPERATION **3-1**

3.1 Overview 3-1
3.2 Message Records 3-2
3.2.1 Reserved 3-3
3.2.2 Activity 3-3
3.2.3 ARINC word 3-4
3.2.4 List Buffer pointer 3-4
3.2.5 Time-tag 3-4
3.2.6 Hit Counter 3-5
3.2.7 Elapsed Time 3-5
3.2.8 Min/Max Elapsed Time 3-5
3.2.9 Decoder Gap 3-5
3.3 Filter Tables 3-5
3.3.1 How Filter Tables work 3-5
3.3.2 Configuring the Filter Tables 3-6
3.4 Transmit Schedules 3-7
3.4.1 How Schedules work 3-7
3.4.2 Creating a Schedule 3-9
3.4.3 Example of a Schedule with explicit timing 3-9
3.5 Sequential Monitor 3-12
3.6 List Buffers 3-13
3.6.1 Receive List Buffers 3-14
3.6.2 Scheduled transmit List Buffers 3-15
3.6.3 Asynchronous transmit List Buffers 3-16

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.7 Error Injection....3-17
3.7.1 Parity errors....3-17
3.7.2 Gap errors....3-17
3.8 Special Events....3-18
3.8.1 Event Log List....3-18
3.8.2 Polling....3-18
3.8.3 Interrupts....3-19

APPENDIX A: FUNCTION REFERENCE A-1

APPENDIX B: MULTI-PROTOCOL/DEVICE PROGRAMS B-1

APPENDIX C: REVISION HISTORY C-1

LIST OF FIGURES

Figure 1.1—A sample CoPilot screen....1-2
Figure 2.1—Example receiver program....2-5
Figure 2.2—Example transmitter program....2-7
Figure 2.3—Example monitor program....2-11
Figure 3.1—Message flow between primary data structures in a typical 429 Device....3-2
Figure 3.2—Message Record structure....3-3
Figure 3.3—Filter Tables....3-6
Figure 3.4—Transmit Schedule....3-7
Figure 3.5—Transmit Schedule for the low-speed timing requirements of Table 3.2....3-10
Figure 3.6—Example with explicit transmit Schedule....3-11
Figure 3.7—Operation of Sequential Record in Interval mode....3-13
Figure 3.8—Receive and transmit List Buffer structure....3-14
Figure 3.9—Operation of an asynchronous transmit List Buffer....3-17
Figure A.1—Standard and reversed label formats of the ARINC 429 word....A-2

LIST OF TABLES

Table 2.1—Example messages....2-2
Table 2.2—Transmit intervals....2-7
Table 2.3—Sequential Monitor filtering options....2-9
Table 3.1—Command Blocks in the transmit Schedule....3-8
Table 3.2—Example transmit intervals....3-9
Table A.1—Protocol-independent BTICard_ functions....A-3
Table A.2—ARINC 429 BTI429_ functions....A-5
Table A.3—Devices grouped by generation and functionality....A-7

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

1. INTRODUCTION

This manual documents the general functions and ARINC 429 functions of Ballard Technology's unified API library called BTIDriver. You can use this manual to learn how to create custom applications for any of Ballard's ARINC 429 BTIDriver-compliant interface products.

1.1 ARINC 429 Overview

ARINC 429 is the specification that defines a local area network used on commercial aircraft and is the industry's standard for transfer of digital data between avionics system elements. The specification describes how an avionics system transmits information over a single twisted and shielded pair of wires (the databus) to as many as 20 receivers connected to that databus. Bidirectional data flow on a given bus is not permitted.

1.2 BTIDriver for ARINC 429

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 to facilitate avionics 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 different avionics databus protocols and different 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 ARINC 429 software applications. Other protocols are documented separately. Appendix B provides guidance for writing multi-protocol BTIDriver applications.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

1.3 CoPilot

As an alternative to writing your own custom programs, you can operate your ARINC 429 interface Device with CoPilot, Ballard Technology's Windows-based GUI (Graphical User Interface) software (see Figure 1.1). CoPilot greatly simplifies such tasks as defining and scheduling bus messages and capturing and analyzing data. Using CoPilot, you can transmit, monitor, and record databus messages with a few clicks of the mouse. CoPilot is user-friendly and has many timesaving features. For example, bus messages can be automatically detected, posted in the hardware tree, and associated with the appropriate attributes from the database of equipment, message, and engineering unit specifications.

CoPilot users can quickly configure, run, and display the activity of multiple databuses in a unified view. Data can be observed and changed in engineering units while the bus is running. The Strip View graphically illustrates the history of the selected data values. Data can also be entered and viewed as virtual instruments (knobs, dials, gauges, etc.) that can be created by the user or automatically generated by dragging and dropping an item into the Control View window.

CoPilot can host multiple channels and databus protocols in the same project, making it an ideal tool for operating multi-protocol Devices. CoPilot can be purchased separately or with a BTIDriver-compliant hardware Device. For more information or a free evaluation copy, call Ballard at (800) 829-1553. In addition, you can learn more about the latest version of CoPilot at www.ballardtech.com.

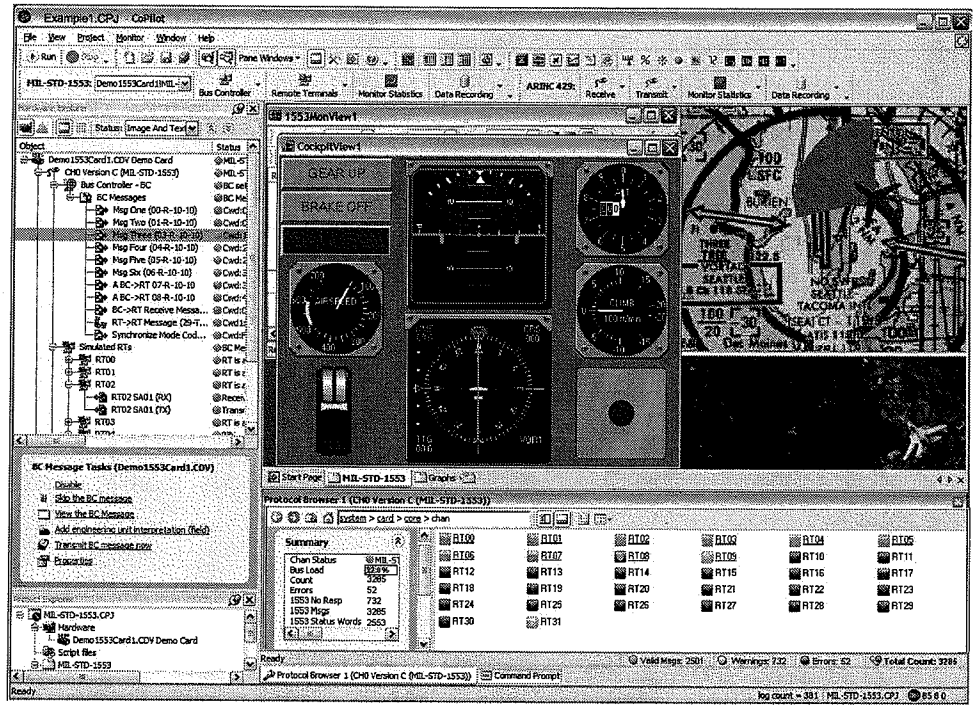


Figure 1.1—A sample CoPilot screen

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

1.4 How to Use This Manual

This manual is designed to be both a tutorial and a reference guide. You can read only the sections that you need and refer to the rest as required. After reading Chapter 2 (Programming Basics) and referring to Appendix A (Function Reference), you should be able to write simple computer programs to operate your Ballard ARINC 429 interface device. Refer to Chapter 3 (Advanced Operation) for more complex applications. This guide can be used in conjunction with the programming manuals for other avionics databus protocols (see Appendix B).

This manual assumes that you are familiar with the essentials of compiling, linking, and running programs in C. It is also assumes that you are familiar with the ARINC 429 protocol. With minor exceptions, the content of this manual also applies to other programming languages.

The following conventions are observed throughout this manual:

- "Device" with a capital "D" is used generically to mean any Ballard ARINC 429 BTIDriver-compliant interface device.
- Driver function names are in bold type and are all prefixed by "**`BTICard_`**" or "**`BTI429_`**" (e.g., **`BTI429_ChConfig`**).
- A small "h" suffix indicates hexadecimal values (e.g., F01Ch)
- Constants defined in the driver software are written in all capital letters (e.g., CHCFG429_DEFAULT).
- 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., **`BTI429_MsgData`**?? to represent **`BTI429_MsgDataRd`** and **`BTI429_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 register your product so that we can keep you informed of updates, customer services, and new product information.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This page intentionally blank.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

2. PROGRAMMING BASICS

This chapter illustrates basic ARINC 429 operation using the BTIDriver library. Examples demonstrate

- operation of ARINC 429 receive channels
- operation of ARINC 429 transmit channels
- monitoring and selectively recording ARINC 429 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 "**`BTI429_`**" for ARINC 429-specific functions. The examples are given with the assumption that CH0 is a receive channel and CH4 is a transmit channel. The transmit and receive channels for your ARINC 429 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 BTIDriver functions for ARINC 429 may be found in Appendix A.

2.1 Terminology

The basic unit of information defined by the ARINC 429 specification is a 32-bit word made up of several bit fields. The most important bit fields are the label (8 bits), SDI (2 bits), and data (variable length). Given the source of the data, the label determines how the data field is interpreted, including

- data format (binary, BCD, ASCII character, etc.)
- data type (temperature, pressure, etc.)
- units (Newtons, degrees Celsius, etc.)

The SDI (Source/Destination Identifier) identifies the source of a word when more than one source exists on the aircraft (e.g., left engine temperature versus right engine temperature). Often, however, the SDI bits are ignored.

Although a 32-bit ARINC 429 word is sometimes referred to as a "label" in the avionics industry, to avoid ambiguity with the 8-bit label field this manual uses the term "message."

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

2.2 Getting Started

The first step in developing an application is identifying the ARINC 429 messages to be handled. For the examples in this chapter, suppose we want to simultaneously

- receive selected messages from a Global Positioning System (GPS)
- simulate transmission of messages from an Air Data System (ADS)
- record all above activity to disk for later analysis

Specifically, assume that among all messages that GPS and ADS units can transmit, we are interested only in those shown in Table 2.1.

Equipment	ARINC 429 Word Name	Octal Label
GPS	Present Position—Latitude	310
GPS	Present Position—Longitude	311
ADS	Computed Airspeed	206
ADS	Total Air Temperature	211
ADS	Altitude Rate	212

Table 2.1—Example messages

Part of the configuration process is associating messages with Message Records. Each Message Record contains a single 32-bit ARINC 429 word, and possibly, some extra data related to that message (e.g., the time-tag). When the Device receives a given message, the full 32-bit word is stored in a designated Message Record. When the Device transmits a given message, the word to be transmitted is retrieved from a predetermined Message Record. More detailed information on Message Records may be found in Section 3.2.

2.3 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 eight general steps, most 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:** Software gains access to the Device hardware through the host computer's operating system. The **`BTICard_CardOpen`** and **`BTICard_CoreOpen`** functions request access and obtain "handles" by which subsequent API functions require (see Section 2.4 for more information on card and core handles).
2. **Reset:** Although not required, it is a good idea to reset any previous hardware configuration of the Device by calling the **`BTICard_CardReset`** function for each core.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3. **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 speed are also set at this point.
4. **Initialize data:** Data associated with transmit messages should be initialized before the Device is activated. Initialization prevents the transmission of invalid messages.
5. **Activate:** **`BTICard_CardStart`** activates all configured channels of a core simultaneously. It should be called separately for each core. Once activated, the Device transmits and/or receives from the databuses independently of the host computer.
6. **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. API functions are provided to simplify data exchange between the host and the Device. Additionally, a library of utility functions is provided to insert and extract the bit fields of an ARINC 429 word. These functions may be used independently of the Device.
7. **Deactivate:** **`BTICard_CardStop`** deactivates a core on the Device. Unless a core is explicitly deactivated, it continues operating even if the application software halts.
8. **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.

2.4 Handles, Cards, and Cores

In order for software to address hardware, the software must be able to specify the hardware so that it is differentiated from other similar, and even dissimilar hardware. Also, if the hardware has more than one section, the software must be able to specify which section. Ballard hardware Devices are implemented with one or more sections called cores. Typically, each core has its own protocol processing circuitry. The BTIDriver functions specify the Device with a card handle and the core with a core handle.

Nearly all functions require a handle parameter. The handle is always the last parameter in any function that requires it. **`BTICard_CardOpen`**, which is always the first function called, obtains a card handle (*hCard*). The card handle uniquely identifies a Device when more than one is used in a single computer. The card handle is different from the card number assigned by the host computer operating system.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

A multi-core Device can have cores that work with the same or different protocols. In a way, the software sees each core as a separate card, so a handle (*hCore*) is needed to differentiate the cores. When passed a card handle (*hCard*) and the core number (*corenum*), **BTICard_CoreOpen** returns a core handle (*hCore*), which uniquely identifies the core/card combination. Therefore, **BTICard_CardOpen** must be called first to provide the card handle for **BTICard_CoreOpen**.

The following are handle-related guidelines to use when developing programs for BTIDriver-compliant Devices:

1. Find the Devices and the number and configuration of their cores by running the test program provided on the distribution disk with the Device (e.g., on Windows® run BTITST32.EXE). When there is only one Device in the system, its *cardnum* is zero (0). When a Device has only one core, its *corenum* is zero (0). At any time, you can use this test program or the Windows Device Manager (for Windows operating systems) to reassign the card number.
2. At the beginning of the program, get the card handle (*hCard*) for each Device by calling **BTICard_CardOpen**.
3. Get the core handle (*hCore*) for each core on each Device by calling **BTICard_CoreOpen**.
4. Use *hCore* as the handle in all subsequent functions except **BTICard_CardClose**.
5. At the end of your program, call **BTICard_CardClose** once for each Device using its *hCard* handle to release the card and all of its core resources.
6. It is advisable to use the above procedure (using both *hCard* and *hCore*) even when the Device has only one core. This makes the code more easily adapted for use with multi-core Devices.
7. Use of *hCard* in place of *hCore* addresses the first core (i.e., Core A with *corenum = 0*). This allows software written for legacy BTIDriver-compliant Devices to operate on the first core of multi-core Devices.
8. Legacy BTIDriver-compliant Devices that had not referred to cores in the documentation have only one core and should use core number (*corenum*) zero (0) in **BTICard_CoreOpen**.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

The following examples show how easy it is to perform these steps using the BTIDriver functions.

2.5 Receiver Example

In this section, we describe an example program that receives latitude and longitude messages (see Table 2.1). The primary task in configuring receive channels is telling the Device which messages it should store in specific Message Records. 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 Record 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 2.1 configures the Device to receive all traffic. The two GPS messages of interest are stored in separate Message Records. The Device writes all other messages to the default record.

```
HCARD hCard;
HCORE hCore;
MSGSTRUCT429 msgdefault, latitude, longitude;
INT cardnum = 0;              //Assumes only one Device has been installed
INT corenum = 0;                      //Assumes only one core on the Device

BTICard_CardOpen(&hCard,cardnum);                              //Open the Device
BTICard_CoreOpen(&hCore,corenum,hCard);                         //Open each core
BTICard_CardReset(hCore);                                      //Reset each core

BTI429_ChConfig(CHCFG429_AUTOSPEED,CH0,hCore);               //Configure channel

                                                               // Define filters
msgdefault.addr = BTI429_FilterDefault(MSGCRT429_DEFAULT,CH0,hCore);
latitude.addr  = BTI429_FilterSet(MSGCRT429_DEFAULT,0310,SDIALL,CH0,hCore);
longitude.addr = BTI429_FilterSet(MSGCRT429_DEFAULT,0311,SDIALL,CH0,hCore);

BTICard_CardStart(hCore);                                      //Start the core

while (!done)                          //Read data as required by application
{
    msgdefault.data = BTI429_MsgDataRd(msgdefault.addr, hCore);
    latitude.data   = BTI429_MsgDataRd(latitude.addr, hCore);
    longitude.data  = BTI429_MsgDataRd(longitude.addr, hCore);
}

BTICard_CardStop(hCore);                                       //Stop each core

BTICard_CardClose(hCard);                                     //Close the Device
```

Figure 2.1—Example receiver program

The code starts by creating message structures named *msgdefault*, *latitude*, and *longitude*. These structures have members for the message address and the value of the data.

BTICard_CardOpen and **BTICard_CoreOpen** are always the first driver functions called since they obtain the handles used by subsequent functions. This example assumes that there is only one Device with a single core installed in the system, so the card number (*cardnum*) and core number (*corenum*) are given values of zero. **BTICard_CardReset** clears the core memory, removing any existing configuration data.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BTI429_ChConfig sets up an empty Filter Table for a receive channel and enables or disables selected options for that channel. This function enables the automatic speed detection feature and selects all other default options. Constants can be included to turn on specific options. Note that this function does not activate the channel. CH0 is a predefined constant referring to physical channel 0 of the core.

A Filter Table is an array of pointers to Message Records. There is one pointer for every possible message type (i.e., every possible label/SDI 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 **BTI429_FilterDefault** function fills the Filter Table with pointers so all messages received on that channel are written to the default record. If **BTI429_FilterDefault** is used, it must precede any calls to **BTI429_FilterSet**. The two **BTI429_FilterSet** functions assign receive messages to Message Records by placing entries in the Filter Table. The **BTI429_FilterDefault** and **BTI429_FilterSet** functions return the address of the Message Record.

The SDIALL constant tells the Device to accept all messages with the given label and any SDI. Different constants may be used to specify that only messages with specific SDIs are to be received. Messages with different SDIs 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. **BTI429_MsgDataRd** may be called at any time to return the full 32-bit ARINC word from a specified Message Record. A library of functions is provided to extract the relevant bit fields from the ARINC word.

The program ends with the required **BTICard_CardStop** and **BTICard_CardClose** functions. Receiving messages on other receive channels only requires extra **BTI429_ChConfig**, **BTI429_FilterDefault**, and **BTI429_FilterSet** function calls.

2.6 Transmitter Example

In this section, we describe an example program that transmits airspeed, temperature, and altitude rate messages (see Table 2.2). Normally, ARINC 429 sources transmit messages periodically at repetition rates prescribed by the ARINC 429 specification. The specification defines a minimum and maximum transmit interval for each message. Many different transmit intervals are called out in the specification, so transmitting many different messages may involve complex timing requirements. The BTIDriver functions handle timing requirements automatically.

To illustrate, recall that this example is simulating an ADS (Air Data System) transmitting three messages. Their transmit intervals are shown in Table 2.2. The code in Figure 2.2 configures the Device to transmit these messages with proper timing.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Word	Octal Label	Minimum Transmit Interval	Maximum Transmit Interval
Computed Airspeed	206	62.5 ms	125 ms
Total Air Temperature	211	250 ms	500 ms
Altitude Rate	212	31.3 ms	62.5 ms

Table 2.2—Transmit intervals

This example starts out similar to the previous one. We first define message structures for the messages of interest. Three arrays to be used for building a Schedule are also created. As always, the **BTICard_CardOpen** and **BTICard_CoreOpen** functions must be called first to obtain the necessary handles. **BTI429_ChConfig** automatically determines that CH4 is a transmit channel and configures it accordingly.

```
HCARD hCard;
HCORE hCore;
MSGSTRUCT429 comp_airspeed, tot_air_temp, altitude_rate;
MSGADDR msgaddr[3];
INT min_intrvls[3];
INT max_intrvls[3];
INT cardnum = 0;              //Assumes only one Device has been installed
INT corenum = 0;              //Assumes only one core on the Device


BTICard_CardOpen(&hCard,cardnum);                          //Open the Device
BTICard_CoreOpen(&hCore,corenum,hCard);                     //Open each core
BTICard_CardReset(hCore);                                  //Reset each core

BTI429_ChConfig(CHCFG429_HIGHSPEED,CH4,hCore);           //Configure channel

                                                           //Create 3 messages
comp_airspeed.addr = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);
tot_air_temp.addr  = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);
altitude_rate.addr = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);

msgaddr[0] = comp_airspeed.addr;     //Set up arrays and transmit intervals
min_intrvl[0] = 63;
max_intrvl[0] = 125;

msgaddr[1] = tot_air_temp.addr;
min_intrvl[1] = 250;
max_intrvl[1] = 500;

msgaddr[2] = altitude_rate.addr;
min_intrvl[2] = 32;
max_intrvl[2] = 62;

BTI429_SchedBuild(3,msgaddr,min_intrvl,max_intrvl,CH4,hCore);//Build Schedule

BTI429_MsgDataWr(0206,comp_airspeed.addr,hCore);          //Initialize data
BTI429_MsgDataWr(0211,tot_air_temp.addr,hCore);
BTI429_MsgDataWr(0212,altitude_rate.addr,hCore);

BTICard_CardStart(hCore);                                  //Start the core

while (!done)                  //Update data as required by the application
{
    BTI429_MsgDataWr(comp_airspeed.data,comp_airspeed.addr,hCore);
    BTI429_MsgDataWr(tot_air_temp.data,tot_air_temp.addr,hCore);
    BTI429_MsgDataWr(altitude_rate.data,altitude_rate.addr,hCore);
}

BTICard_CardStop(hCore);                                   //Stop each core

BTICard_CardClose(hCard);                                 //Close the Device
```

Figure 2.2—Example transmitter program

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

The easiest way to create a Schedule that transmits the three messages at their proper transmit intervals is to use the **BTI429_SchedBuild** function. To do this, we first create the three messages using **BTI429_MsgCreate**, which returns their addresses. **BTI429_MsgCreate** in transmit is equivalent to **BTI429_FilterSet** in receive. Next, we set the values of three arrays: one for the message addresses, one for minimum transmit intervals, and one for maximum transmit intervals. Information on a given message is contained in the same position in each of these arrays. The minimum and maximum transmit intervals (rounded to integer values in milliseconds) are defined as specified in Table 2.2.

The **BTI429_SchedBuild** function constructs a Schedule in the Device memory. 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 **BTI429_SchedBuild** is to schedule transmission of three messages on CH4 from the information in the three arrays. An alternative to using **BTI429_SchedBuild** is to explicitly define your own Schedule as described in Section 3.4.

The Message Records are initialized using **BTI429_MsgDataWr** before **BTICard_CardStart** commands the Device to begin transmitting. Here we initialized all three messages to zero except for their labels. Notice that the label is the least significant part of the data and that it is entered in octal. See Appendix A for more information regarding the order of bits in an ARINC word. Also using **BTI429_MsgDataWr**, we can update the data value of a message at any time.

The three messages are repeatedly transmitted at the proper rates until the core is halted by **BTICard_CardStop**. As always, the program ends with **BTICard_CardClose**.

2.7 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 messages which is useful for analyzing and reconstructing all or selected bus activity. This activity is stored in what is called a Sequential Record. Additional information on the Sequential Monitor may be found in Section 3.5.

The code in Figure 2.3 combines the previous receive and transmit examples while configuring the Sequential Monitor to capture only the five message types defined in the examples. Specifically, the Sequential Monitor will capture only the latitude and longitude messages from the receive channel and all words transmitted by the core.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Use these functions and parameters → to record ↓	BTI429_ChConfig	Receive BTI429_FilterSet	Transmit BTI429_MsgCreate
from all ARINC 429 messages on selected channels	CHCFG429_SEQALL	don't care	don't care
from selected 429 messages on selected channels	CHCFG429_SEQSEL	MSGCRT429_SEQ	MSGCRT429_SEQ

Table 2.3—Sequential Monitor filtering options

Filtering irrelevant messages out of the bus traffic conserves Sequential Monitor memory and makes the Sequential Record easier to analyze. Filtering for the Sequential Monitor can be established at either the channel level or the message level. If the CHCFG429_SEQALL constant is used in **`BTI429_ChConfig`**, then all ARINC 429 messages on that channel are saved in the Sequential Record. If only specific messages are to be saved then the MSGCRT429_SEQ constant is used in either **`BTI429_FilterSet`** for receive or **`BTI429_MsgCreate`** for transmit. These are summarized in Table 2.3.

Much of the code in Figure 2.3 is a combination of modified fragments from the previous examples. Since we want to monitor all transmitted messages, the CHCFG429_SEQALL constant is used in place of the CHCFG429_DEFAULT constant in **`BTI429_ChConfig`** for CH4. To save the latitude and longitude messages in the Sequential Record, the MSGCRT429_SEQ constant is used in their respective **`BTI429_FilterSet`** functions. **`BTICard_SeqConfig`** allocates memory for and configures the Sequential Monitor. The SEQCFG_DEFAULT constant selects all default settings. **`BTICard_CardStart`** activates all configured channels on the core. After activation, the core 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 ARINC 429 message value and its time-tag. **`BTICard_SeqRd`** copies the available records from the onboard 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_SeqFindNext429`** find and point (*pRec429*) to the next occurrence of a 429-type record. The desired values are then printed from the record. 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`**.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
HCARD hCard;
HCORE hCore;
MSGSTRUCT429 msgdefault, latitude, longitude;
MSGSTRUCT429 comp_airspeed, tot_air_temp, altitude_rate;
MSGADDR msgaddr[3];
INT min_intrvls[3];
INT max_intrvls[3];
USHORT seqbuf[2048];
ULONG seqcount;
SEQFINDINFO sfinfo;
LPSEQRECORD429 pRec429;
INT cardnum = 0;              //Assumes only one Device has been installed
INT corenum = 0;              //Assumes only one core on the Device

BTICard_CardOpen(&hCard,cardnum);                          //Open the Device
BTICard_CoreOpen(&hCore,corenum,hCard);                     //Open each core
BTICard_CardReset(hCore);                                  //Reset each core

BTI429_ChConfig(CHCFG429_AUTOSPEED,CH0,hCore);  //Configure receive channel

BTI429_ChConfig(CHCFG429_SEQALL,CH4,hCore);    //Configure transmit channel

//Create 3 transmit messages
comp_airspeed.addr = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);
tot_air_temp.addr  = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);
altitude_rate.addr = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);

msgaddr[0] = comp_airspeed.addr;     //Set up arrays and transmit intervals
min_intrvl[0] = 63;
max_intrvl[0] = 125;

msgaddr[1] = tot_air_temp.addr;
min_intrvl[1] = 250;
max_intrvl[1] = 500;

msgaddr[2] = altitude_rate.addr;
min_intrvl[2] = 32;
max_intrvl[2] = 62;

BTI429_SchedBuild(3,msgaddr,min_intrvl,max_intrvl,CH4,hCore);//Build Schedule

BTI429_MsgDataWr(0206,comp_airspeed.addr,hCore);          //Initialize data
BTI429_MsgDataWr(0211,tot_air_temp.addr,hCore);
BTI429_MsgDataWr(0212,altitude_rate.addr,hCore);

//Define filters
msgdefault.addr = BTI429_FilterDefault(MSGCRT429_DEFAULT,CH0,hCore);
latitude.addr   = BTI429_FilterSet(MSGCRT429_SEQ,0310,SDIALL,CH0,hCore);
longitude.addr  = BTI429_FilterSet(MSGCRT429_SEQ,0311,SDIALL,CH0,hCore);

BTICard_SeqConfig(SEQCFG_DEFAULT,hCore);  //Configure the Sequential Record

BTICard_CardStart(hCore);                                  //Start the core

while (!done)
{
    seqcount = BTICard_SeqRd(seqbuf,hCore);     //Read the Sequential Record
    BTICard_SeqFindInit(seqbuf,seqcount,&sfinfo); //Initialize Find
functions
    while(!BTICard_SeqFindNext429(&pRec429,&sfinfo))//Find/process 429 records
    {
        //Write to disk, display data, etc. as desired. For example:

        printf("\nData:%08lX Time Stamp:%08lX",
               pRec429->data,
               pRec429->timestamp);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
    }

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

BTICard_CardStop(hCore);                          //Stop each core
BTICard_CardClose(hCard);                        //Close the Device
```

Figure 2.3—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 may be found on the distribution disk and detailed function descriptions are found in Appendix A. Further information may be contained in the README files on the distribution disk.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This page intentionally blank.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3. 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.

3.1 Overview

The unique power of your BTIDriver Device comes from the on-board resources that implement many of the Device capabilities, thereby freeing the host processor to perform other tasks. The speed of these resources allow the Device to simultaneously process all ARINC 429 channel features including transmit scheduling, receive filtering, and monitoring of bus activity. Additionally, the Device manages the interface to the host, arbitrating access to the on-board memory, and provides low-level ARINC 429 encoding and decoding. The on-board memory contains all configuration data structures described in the following sections.

All exchanges between the host and ARINC 429 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 ARINC 429 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 API functions.

Figure 3.1 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.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

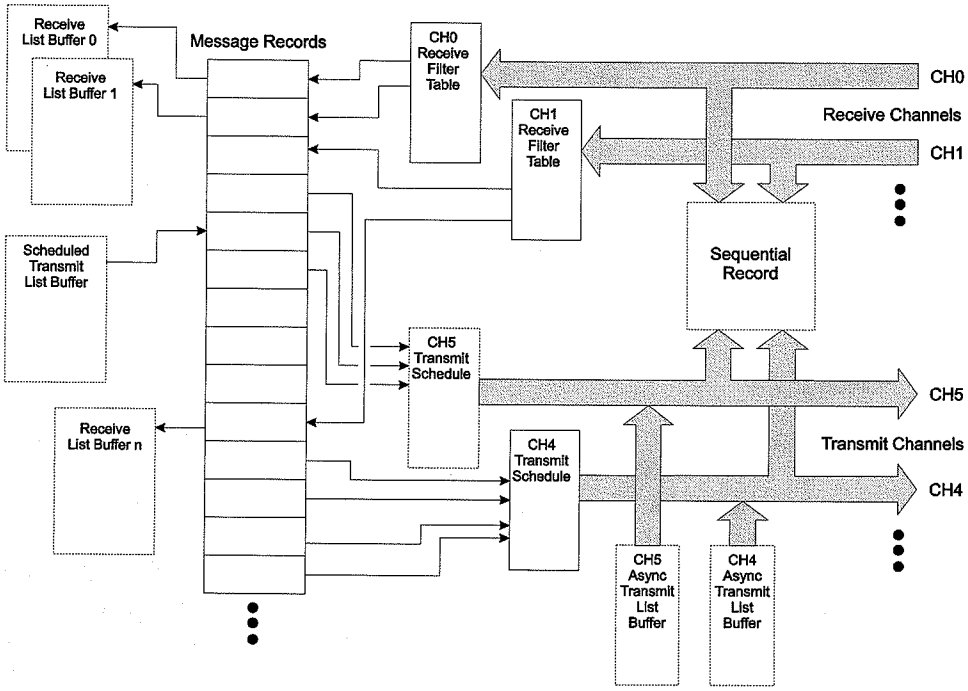


Figure 3.1—Message flow between primary data structures in a typical 429 Device

3.2 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 2. A Message Record structure (see Figure 3.2) 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, and Message fields are used. The Device updates the fields continuously while it is activated (i.e., between **`BTICard_CardStart`** and **`BTICard_CardStop`**). The rest of this section describes these data fields. (Note that some options are mutually exclusive.) Sections 3.3 and 3.4 describe how the receive and transmit channels use Message Records.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

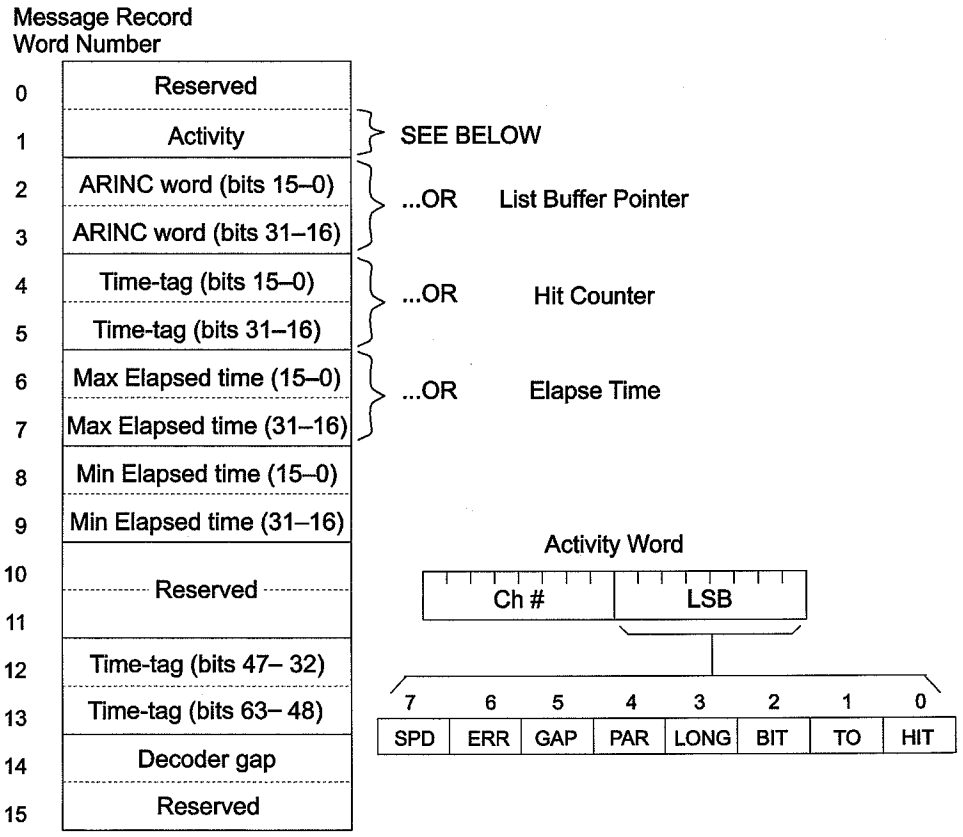


Figure 3.2—Message Record structure

3.2.1 Reserved

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

3.2.2 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 Device senses the channel and automatically fills in this value, which is in the range of 0 to 255.

SPD: This bit indicates the speed of the channel. A zero indicates low speed (12.5 Kbps) and a one indicates high speed (100 Kbps).

ERR: This bit indicates an error was detected in a word, and only has meaning for received words. It is the logical OR of GAP, PAR, LONG, BIT, and TO.

GAP: Indicates that the word was not preceded by a gap of at least four bit times. This bit only has meaning for received words.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

PAR: Indicates that the word was received with a parity error. This bit only has meaning for received words.

LONG: Indicates that the word was received with more than 32 bits, of which only 32 bits are represented in the Message field. This bit only has meaning for received words.

BIT: Indicates that the word was received with some kind of timing error in at least one bit. This bit only has meaning for received words.

TO: Indicates that a time-out occurred, probably caused by a short word (less than 32 bits) or a noise burst that looked like the start of a word. This bit only has meaning for received words.

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 **BTI429_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 **BTI429_MsgIsAccessed**. This bit has meaning for both received and transmitted words.

3.2.3 ARINC word

The ARINC word is the 32-bit data value of the ARINC 429 message. Bits seven through zero contain the label. See Appendix A for a complete interpretation of an ARINC 429 message.

3.2.4 List Buffer pointer

When a List Buffer is associated with a Message Record, the ARINC word field is replaced with a pointer to the List Buffer. See Section 3.6 for more information on List Buffers.

3.2.5 Time-tag

The time-tag is a 64-bit value derived from an internal clock. The resolution and range of the time-tag is dependent on the Device and its capabilities. For example, some Devices can be configured to use binary time values or BCD IRIG based time values (see **BTICard_TimerStatus** for more information). Also, on some Devices the resolution of time-tag values may be adjusted with the **BTICard_TimerResolution** function. Please see the **BTICard_Timer??** and **BTICard_IRIG??** function synopses in Appendix A for more information on Device dependencies.

For some Devices (BUSBox BB1xxx series Devices and OmniBus series Devices when using the binary timer), the time-tag of a transmitted word represents when the word is loaded into the encoder, *not* when the word is actually transmitted. Similarly, the time-tag of a received word represents when the word was read from the decoder. Thus, the time-tag can deviate slightly from the actual time of the ARINC 429 bus activity.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.2.6 Hit Counter

This number is incremented every time the Message Record is accessed by the Device, so it represents the number of times a message has been received or transmitted. This option may be enabled for all messages on the channel by **`BTI429_ChConfig`** or for a single message by **`BTI429_MsgCreate`**, **`BTI429_FilterDefault`**, and **`BTI429_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.

3.2.7 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 Device 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.

3.2.8 Min/Max Elapsed Time

The Minimum and Maximum Elapsed Times are the worst case Elapsed Times since the Device was last activated. The Device calculates the Elapsed Time, and if it is not between the current minimum and maximum, the appropriate value is updated. The Elapsed Time and Min/Max Elapsed Time options are enabled by the **`BTI429_ChConfig`** function. This function also initializes the minimum time and the maximum time to zero.

3.2.9 Decoder Gap

For Message Records associated with a receive channel, the decoded inter-word gap time preceding the current message is saved. This gap time has a resolution of one-half bit times, and has a maximum range of eight bit times. This is useful for analyzing the measured gap time if a decoder GAP error occurs (as discussed in Section 3.2.2).

3.3 Filter Tables

Each receive channel has a Filter Table, as shown in Figure 3.3. A Filter Table is an array of pointers to Message Records. It contains one pointer for every label/SDI combination (256 x 4 pointers). The label/SDI bits of an ARINC word form an index into the Filter Table.

3.3.1 How Filter Tables work

When a word is received from the ARINC 429 databus, the Device examines the label and SDI bits and retrieves the corresponding pointer from the Filter Table. If the pointer is zero, the Device quits processing the word. Otherwise, the Device writes the word to the Message Record to which the Filter Table points for that label/SDI. Other processing may follow depending on which options are enabled. For example, if the Elapsed Time option is enabled, the Device calcu-

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

lates the time elapsed since the last reception of the word and writes this to the Message Record.

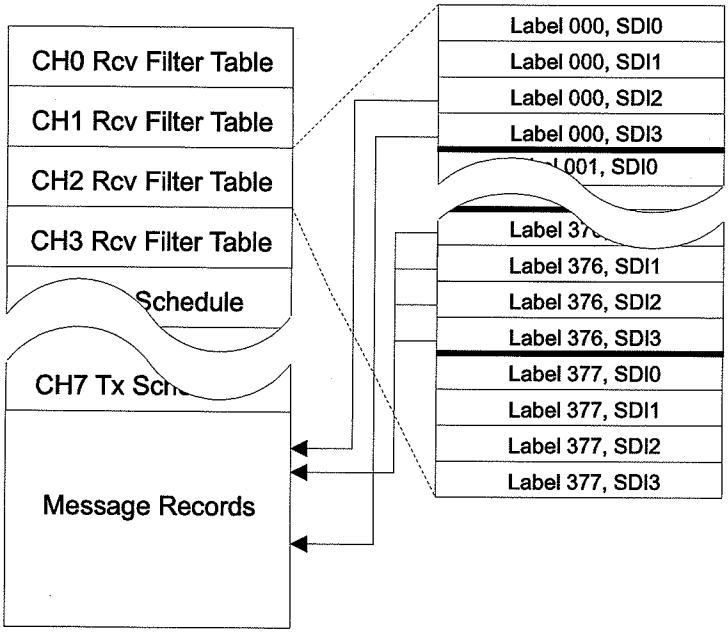


Figure 3.3—Filter Tables

3.3.2 Configuring the Filter Tables

The user sets Filter Table entries to point to Message Records with the **BTI-429_FilterSet** and **BTI429_FilterDefault** functions. **BTI429_Ch-Config** sets up the Filter Table and therefore must precede any calls to **BTI429_FilterSet** or **BTI429_FilterDefault**. Any number of Filter Table entries may point to the same Message Record. For example, the SDIALL constant used in the **BTI429_FilterSet** function sets the pointers for all four SDIs of a particular label to the same Message Record, as shown for label 376 in Figure 3.3.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.4 Transmit Schedules

A Schedule controls the transmission of words onto the ARINC 429 databus and is executed by the Device firmware. There are two ways to create a Schedule. Chapter 2 demonstrated the easiest way: using the **`BTI429_SchedBuild`** function to automatically construct a Schedule. This section describes the Schedule in more detail and tells how to explicitly construct a Schedule.

3.4.1 How Schedules work

The Schedule must be loaded by the host processor 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 3.4. Each Command Block contains one of the opcodes shown in Table 3.1.

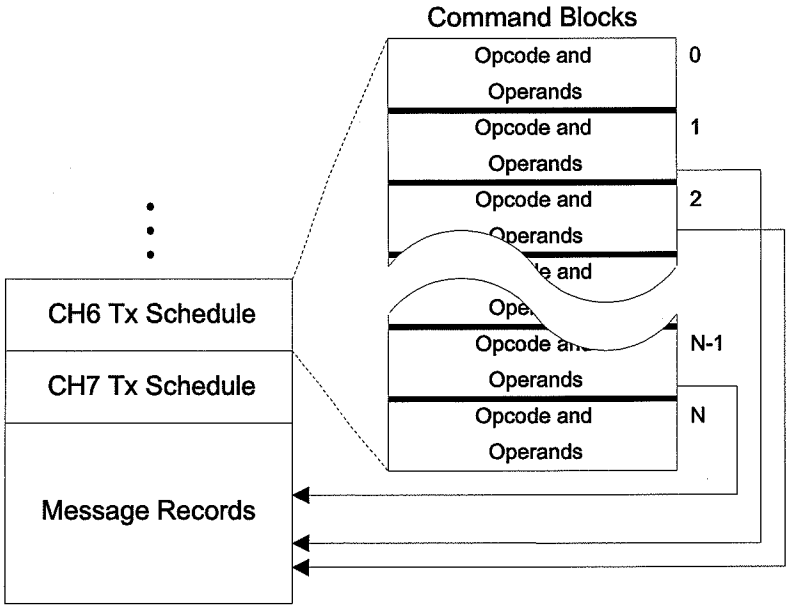


Figure 3.4—Transmit Schedule

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Opcode Name	Function	Description
MESSAGE	BTI429_SchedMsg	Transmits the ARINC 429 word from the Message Record
GAP	BTI429_SchedGap	Inserts a timed gap between transmissions (allows asynch msgs)
ENTRY	BTI429_SchedEntry	Indicates the starting point for the Schedule
HALT	BTI429_SchedHalt	Halts the Schedule
PAUSE	BTI429_SchedPause	Halts processing of Schedule until BTI429_ChResume is called
RESTART	BTI429_SchedRestart	Restarts the Schedule at the beginning
LOG	BTI429_SchedLog	Generates Event Log List entry (see Section 3.8.1)
BRANCH	BTI429_SchedBranch	Jumps to specified Command Block and resumes execution.
CALL	BTI429_SchedCall	Jumps to specified Cmd Block, saving the return address on stack.
RETURN	BTI429_SchedReturn	Returns to address following last call.
NOP	BTI429_SchedNop	No operation.
PULSE	BTI429_SchedPulse	Pulse an external discrete signal
LISTGAP	BTI429_SchedGapList	Inserts a conditional timed gap for asynchronous messages
FIXEDGAP	BTI429_SchedGapFixed	Inserts a timed gap that does not allow asynchronous messages

Table 3.1—Command Blocks in the transmit Schedule

Though very complex Schedules may be created using the special commands listed in Table 3.1, the typical Schedule consists of a loop of MESSAGE and GAP Command Blocks. When the Device processes a MESSAGE Command Block, it retrieves the ARINC 429 word to be transmitted from the Message Record pointed to by the operand in the Command Block (see Figure 3.4). The Device loads the word 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 triggers transmission of the current contents of the encoder and specifies the period the transmitter must wait before starting another transmission.

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

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

1. A MESSAGE Command Block should be followed by either another MESSAGE Command Block or a GAP Command Block. No transmission of the message occurs until a subsequent MESSAGE or GAP Command Block is processed.

2. 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.

3. If a MESSAGE Command Block follows another MESSAGE Command Block, then a 4 bit time gap is automatically inserted between the two messages.

4. 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).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

5. 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. Subroutines are not supported by all Devices. Please refer to the Device Dependency section of the scheduling functions in Appendix A for details.

3.4.2 Creating a Schedule

The Schedule is programmed explicitly using **BTI429_SchedMsg**, **BTI-429_SchedGap**, and other scheduling functions shown in Table 3.1. 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 ARINC 429 words. The other opcodes control the processing of the Schedule. The details of each scheduling function, as well as its device dependency, is explained in Appendix A. The following section demonstrates the use of the scheduling functions.

3.4.3 Example of a Schedule with explicit timing

The transmit example in Chapter 2 used **BTI429_SchedBuild** to automatically construct the transmit Schedule required to meet given timing requirements. In contrast, this example specifies timing explicitly. Additionally, it verifies the transmitter's timing performance. It configures a channel to receive data from the transmitter through its internal, wraparound, self-test feature and records the minimum and maximum transmit intervals.

For this example, suppose that we want to transmit the two messages shown in Table 3.2. We choose target transmit intervals that are the approximate averages of the minimum and maximum transmit intervals given in the ARINC 429 specification. A transmit Schedule that meets these timing requirements (accounting for word length and interword gaps) is shown in Figure 3.5. If transmission of this sequence of words and gaps is repeated, the timing requirements for each word will be met. At slow speed (12.5 Kbps), a bit time is 0.08 ms, and at high speed (100 Kbps), it is 0.01 ms, so each word takes either 2.56 ms or 0.32 ms.

ARINC Word (octal label)	Minimum Transmit Interval	Maximum Transmit Interval	Target Interval
Computed Airspeed (206)	62.5 ms	125 ms	90 ms
Altitude Rate (212)	31.3 ms	62.5 ms	45 ms

Table 3.2—Example transmit intervals

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

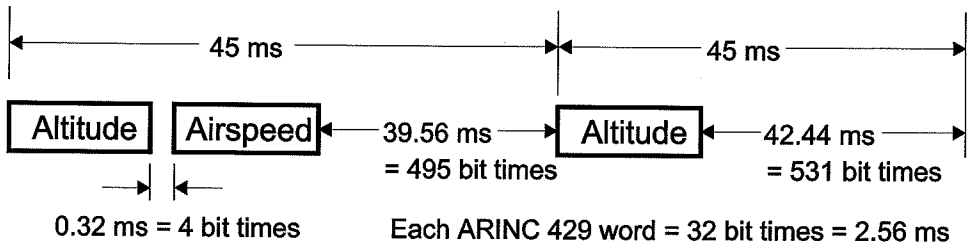


Figure 3.5—Transmit Schedule for the low-speed timing requirements of Table 3.2

The code for this example (Figure 3.6) begins by creating message structures. MSGSTRUCT429 has members for the address of the Message Record and the value of the data; it is used as in previous examples. We will transmit from two Message Records, and we will receive these transmissions into a different pair of Message Records. MSGFIELDS429 is a structure with members for each of the fields in a Message Record. Its use is explained later.

After obtaining the handle of the core with **`BTICard_CardOpen`** and **`BTICard_CoreOpen`** we use **`BTI429_ChConfig`** to configure a receive channel (CH0) with the Min/Max Elapsed Time enabled. The Min/Max Elapsed Time option is explained in Section 3.2.8.

Next, with the **`BTI429_FilterSet`** functions, we allocate Message Records in which to save messages for the given labels. As in the examples of Chapter 2, we are ignoring SDI bits.

The next **`BTI429_ChConfig`** configures a transmit channel (CH4) with the self-test option enabled. The self-test option sends the output from CH4 back into all of the receive channels. Only one transmit channel should be connected to the self-test bus at any given time. **`BTI429_MsgCreate`** is used to create the two messages to be transmitted.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
HCARD hCard;
HCORE hCore;
MSGSTRUCT429 xmt_airspeed, xmt_altitude, rcv_airspeed, rcv_altitude;
MSGFIELDS429 airspeed_record, altitude_record;
INT cardnum = 0;  '            //Assumes only one Device has been installed
INT corenum = 0;                     //Assumes only one core on the Device

BTICard_CardOpen(&hCard,cardnum);                              //Open the Device
BTICard_CoreOpen(&hCore,corenum,hCard);                         //Open each core
BTICard_CardReset(hCore);                                      //Reset each core

//Configure receive channel and filters
BTI429_ChConfig(CHCFG429_MAXMIN,CH0,hCore);
rcv_airspeed.addr =
BTI429_FilterSet(MSGCRT429_DEFAULT,0206,SDIALL,CH0,hCore);
rcv_altitude.addr =
BTI429_FilterSet(MSGCRT429_DEFAULT,0212,SDIALL,CH0,hCore);

//Configure transmit channel and create messages
BTI429_ChConfig(CHCFG429_SELFTEST,CH4,hCore);
xmt_airspeed.addr = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);
xmt_altitude.addr = BTI429_MsgCreate(MSGCRT429_DEFAULT,hCore);

                                              //Create the transmit Schedule
BTI429_SchedMsg(xmt_altitude.addr,CH4,hCore);
BTI429_SchedMsg(xmt_airspeed.addr,CH4,hCore);
BTI429_SchedGap(495,CH4,hCore);
BTI429_SchedMsg(xmt_altitude.addr,CH4,hCore);
BTI429_SchedGap(531,CH4,hCore);

      //Initialize the message records from which the Device will transmit
BTI429_MsgDataWr(0x0000008A,xmt_altitude.addr,hCore);
BTI429_MsgDataWr(0x00000086,xmt_airspeed.addr,hCore);

BTICard_CardStart(hCore);

while (!done)                                 //Read and print min/max times
{
BTI429_MsgBlockRd(&airspeed_record,rcv_airspeed.addr,hCore);
BTI429_MsgBlockRd(&altitude_record,rcv_altitude.addr,hCore);

printf("Altitude max interval: %u \n", altitude_record.maxtime);
printf("Altitude min interval: %u \n", altitude_record.mintime);
printf("Airspeed max interval: %u \n", airspeed_record.maxtime);
printf("Airspeed min interval: %u \n", airspeed_record.mintime);
}

BTICard_CardStop(hCore);                                        //Stop each core

BTICard_CardClose(hCard);                                      //Close the Device
```

Figure 3.6—Example with explicit transmit Schedule

The **BTI429_SchedMsg** and **BTI429_SchedGap** functions create the Schedule using the sequence of messages and gaps shown in Figure 3.5. Notice that the minimum (4-bit) interword gap need not be explicitly scheduled. In addition, the time parameter of the **BTI429_SchedGap** function is in units of bit times. At low speed, a bit time is 0.08 ms.

The **BTI429_MsgDataWr** function initializes the values to be transmitted for altitude and airspeed rate. The octal labels 206 and 212 translate to hexadecimal 86 and 8A, respectively. We do not care what the data values are for this example, so we have zeroed all but the label bits.

After **BTICard_CardStart** commands the core to begin operating, the two messages are transmitted in accordance with the Schedule. Meanwhile, the re-

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ceiver records these transmissions along with the minimum and maximum times between successive receptions of each message.

Within the while loop we use **BTI429_MsgBlockRd** to read each Message Record into the application. A pointer (e.g., *&airspeed_record*) to the destination structure (MSGFIELDS429), which we declared at the beginning of the code, is a parameter of this function. The minimum and maximum elapsed time values are members of that structure as shown in Figure 3.2. The **printf** functions display the measured ranges of the transmit intervals, which should correlate very closely to our expected values.

The core is stopped using **BTICard_CardStop.** The min/max measurements can be read from the core before or after calling **BTICard_CardStop**.

Schedules involving more messages may be implemented the same way, though, as mentioned in Chapter 2, calculating the timing requirements becomes substantially more difficult. Other actions, such as Event Log List entries, may also be scheduled.

3.5 Sequential Monitor

The Sequential Monitor records a time-tagged history of selected ARINC 429 messages transmitted and received by the core. This history is stored in a buffer called the Sequential Record. There is one Sequential Record per core, so even if the core supports multiple channels of the same or different protocols, all messages associated with the core are stored in the one Sequential Record (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 2.6.

By default, recording halts when the on-board Sequential Record is full. This happens in order to prevent unread data from being overwritten when the host gets behind in reading the data from Sequential Record. If the host continues to read from the Sequential Record, then the it will not become full and therefore not halt recording. An alternate mode for the Sequential Monitor is to be configured with the SEQCFG_CONTINUOUS flag in **BTICard_SeqConfig,** in which recording is not halted. In this mode, it automatically wraps around and continues recording, overwriting old messages. This mode is useful to keep a history of the most recently monitored data in the Sequential Record rather than removing it from the Device while it is running. The core 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 core can log an entry on every n^{th} message recorded into the Sequential Record. (See Section 3.8.1 for more information on enabling entries into the Event Log List.)

The Sequential Monitor records data to the Sequential Record only while the core is running (**BTICard_CardStart**). However, while the core is running, the Sequential Monitor can be stopped and restarted without affecting other core functions. **BTICard_SeqStart** is used to start the Sequential Monitor; it also stops and initializes it if necessary before restarting it. **BTICard_SeqStop** stops data from being added to the Sequential Record. If **BTICard_SeqRe-**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

sume 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 core adds a message to the Sequential Record, it updates the head pointer; when the host reads the contents of the Sequential Record, the tail pointer is updated. The Sequential Record may be read using any one of a family of functions: **BTICard_SeqRd** reads a single record at a time, while **BTICard_SeqBlkRd** and **BTICard_SeqCommRd** read as many records as they can (except for BUSBox BB1xxx series Devices). Note that in the special case of BUSBox BB1xxx series Devices, all three functions read as many records as they can and produce similar results. As long as one of these functions is used to read the Sequential Record, the head and tail pointers are automatically maintained.

Two additional options control the contents of the Sequential Monitor: Interval mode and Delta mode. In Interval mode, the core only records the first instance of a message in successive intervals of time (as illustrated in Figure 3.7). In Delta mode, an entry is added to the Sequential Record for a message only when the data changes. Both options are device dependent, and are enabled by including the proper flags in the **BTICard_SeqConfig** function. Please refer to Appendix A for details.

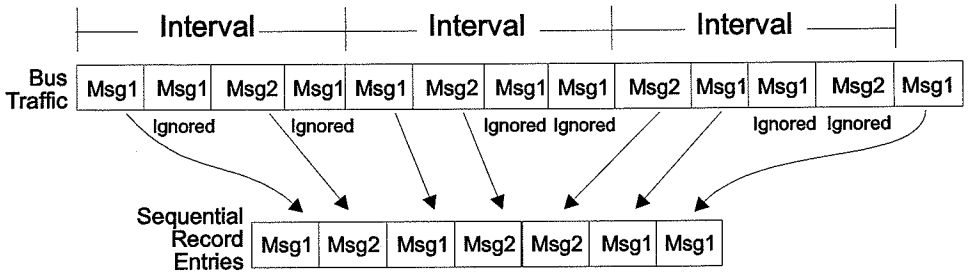


Figure 3.7—Operation of Sequential Record in Interval mode

3.6 List Buffers

By providing a separate location for every label and SDI combination, the Message Record array simplifies retrieving received data and updating transmitted data. However, each Message Record holds only one ARINC word. Alternatively, List Buffers can be used to store sequences of words. Lists Buffers may be used to buffer a changing receive message, to transmit a predefined sequence of words, or to support file transfer protocols such as ARINC 615.

A List Buffer may service messages on either a receive or a transmit channel. When a List Buffer is associated with a message, the ARINC word field in the Message Record (shown in Figure 3.2) is replaced with a pointer to the List Buffer. The different types of List Buffers have the same structure (Figure 3.8), but operate very differently as described in the following sections. Example programs that use list buffering may be found on the distribution disk.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

List Buffers are defined by the user using the **BTI429_ListRcvCreate**, **BTI429_ListXmtCreate**, and **BTI429_ListAsyncCreate** functions. The size of each List Buffer is measured in the number of ARINC words it holds. The number of List Buffers that can be allocated and the size of each are limited only by available Device memory.

3.6.1 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 **BTI429_ListRcvCreate**.

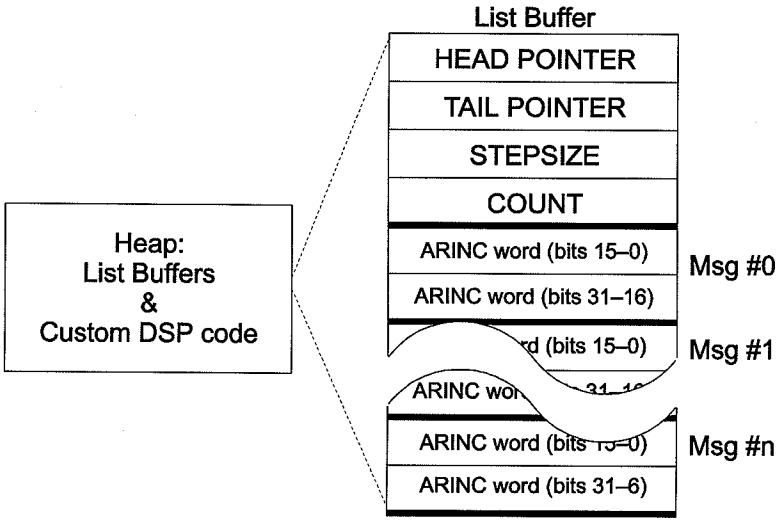


Figure 3.8—Receive and transmit List Buffer structure

FIFO List Buffer:

When the value of received data in a particular label/SDI is rapidly changing and it is important not to lose any of the data, then a FIFO receive List Buffer may be used. This may be especially true in communications or file transfer protocols such as ARINC 615 where there are streams of words with identical label fields separated by minimum (4-bit) gap times. Thus, a Device participating in such a protocol must be capable of buffering long sequences of words.

The Device adds messages to a FIFO List Buffer as they are received, and the user retrieves them sequentially with the **BTI429_ListDataRd** function. The user is not required to maintain the pointers as long as the **BTI429_ListDataRd** function is used to access the buffer. The pointers are automatically changed when a new word is entered into the list (i.e., when it is received) and when a word is read from the list using **BTI429_ListDataRd**. 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.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

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 on certain Devices. 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 **BTI-429_ListDataRd** always reads the most recent complete copy of a received message. Data integrity is guaranteed by some Devices when **BTI429_MsgCommRd** is used, and therefore those Devices do not support ping-pong list buffers. Please refer to **BTI429_ListRcvCreate** in Appendix A for details.

3.6.2 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 **BTI429_ListXmtCreate**. The type is specified using predefined constants. The user writes words sequentially to the List Buffer using the **BTI429_ListDataWr** function. The user is not required to maintain the pointers as long as the **BTI429_ListDataWr** function is used. Transmission timing is controlled by the transmit Schedule. When the Schedule indicates that a message should be transmitted, the next message in the list is used.

FIFO List Buffer:

Whenever the Schedule indicates that a word 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. If messages are not updated fast enough by the host and all words have been transmitted at least once, then the last (most recent) word written by the host to the FIFO List Buffer is the word that is transmitted until another word 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.

Ping-Pong List Buffer:

As in receive, the transmit ping-pong List Buffer guarantees data integrity when the host computer and the Device simultaneously access the same message, which could cause the data being transmitted by the Device to contain part of the old message and part of the new message. With a ping-pong List Buffer, the Device transmits the last complete message loaded using **BTI429_ListDataWr**. Data integrity is guaranteed by some Devices when **BTI429_MsgCommWr** is used, and therefore those Devices do not support ping-pong list buffers. Please refer to **BTI429_ListXmtCreate** in Appendix A for details.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Circular List Buffer:

With a circular List Buffer, transmissions repeatedly loop through the entire list buffer. This feature would greatly simplify 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.

3.6.3 Asynchronous transmit List Buffers

Although there may be other uses, asynchronous transmit List Buffers are intended to support communications protocols such as ARINC 615. Your Device is capable of transmitting the asynchronous bursts of words typically involved in communications protocols without disrupting the timing of periodic words (the regular scheduled messages). A bidirectional communications protocol would involve the use of receive List Buffers and asynchronous transmit List Buffers.

Operation of the asynchronous transmit List Buffer is very simple. When an allocated asynchronous transmit List Buffer contains words that have not been transmitted (i.e., is not empty), the Device automatically transmits words from the asynchronous transmit List Buffer by interleaving them in the gaps of a running Schedule. The Device fills any scheduled gap with as many words as will fit from the asynchronous transmit List Buffer. The gap is then adjusted to preserve the timing of scheduled messages as shown in Figure 3.9. This architecture allows an application program to load conveniently sized packets of data from a file and let the Device automatically manage their transmission. **`BTI-429_ListStatus`** may be used to determine whether a List Buffer is empty.

Unlike other List Buffers, an asynchronous transmit List Buffer is associated with a channel, not a Message Record. To implement an asynchronous transmit List Buffer, use **`BTI429_ListAsyncCreate`** to create the list and associate it with a channel. Then create a transmit Schedule automatically using **`BTI429_SchedBuild`** (Section 2.6) or explicitly using **`BTI429_SchedMsg`** and **`BTI429_SchedGap`** (Section 3.4). If only asynchronous messages are to be transmitted and no messages are to be scheduled, then a dummy Schedule must be created explicitly with a single large gap. Once the Schedule is running, use **`BTI429_ListDataWr`** to pass the data for transmission. The distribution disk has example programs that illustrate the use of asynchronous transmit List Buffers.

Asynchronous transmit List Buffers are FIFO buffers. Words written to the list with **`BTI429_ListDataWr`** are transmitted only once in the order they are written. When the list is empty, no words are transmitted from the list until the host writes new words to it. As with other List Buffers, the user is not required to maintain the pointers as long as **`BTI429_ListDataWr`** is used to add words to the list.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

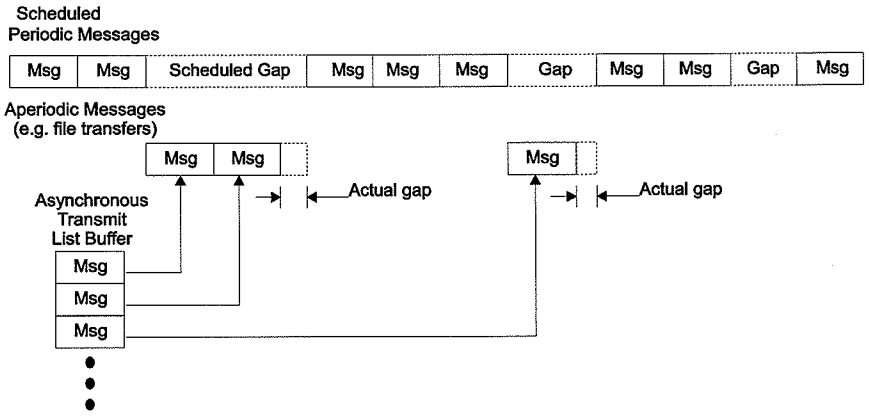


Figure 3.9—Operation of an asynchronous transmit List Buffer

3.7 Error Injection

In some situations, a user may want to evaluate a receiver's response to protocol errors in a transmitted message. For such situations, the user can selectively direct the Device to inject parity errors or gap errors.

3.7.1 Parity errors

ARINC 429 specifies that messages (32-bit words) should be transmitted with odd parity, so the Device defaults to odd parity. However, any transmit or receive channel on the Device can be configured using **`BTI429_ChConfig`** so the parity is odd, even, or used as data. When used as data, the parity circuits do not generate or check parity. Also, transmit channels and transmit messages can be made to invert the parity bit (i.e., send a parity error) using other constants in **`BTI429_ChConfig`** and **`BTI429_MsgCreate`**. Please refer to the function descriptions in Appendix A for the appropriate constants to set these options.

3.7.2 Gap errors

According to the ARINC 429 specification, the gap between messages should be at least four (4) bit times. This requirement is automatically met when transmit Schedules are created using **`BTI429_SchedBuild`** or the explicit scheduling described in Section 3.4. Normally, two sequential messages in a Schedule are automatically separated by a four bit time gap. If a gap of less than four bit times is desired, then an explicit Schedule must be created using **`BTI429_SchedMsgEx`** instead of **`BTI429_SchedMsg`** for the message preceding the short gap. A parameter in **`BTI429_SchedMsgEx`** specifies the length of the gap.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.8 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 for each core. 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 ARINC 429 related activity. See the description of **BTICard_EventLogRd** in Appendix A for a table of event types for different protocols.

3.8.1 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_EventLogConfig** function creates the Event Log List for a core. 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.

3.8.2 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.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.8.3 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_IntInstall** function is used to enable the hardware interrupt and to associate the interrupt service routine with the interrupts from a core. 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. More discussion may be found under **BTICard_IntInstall** in Appendix A. Also, see the interrupt examples on the software distribution disk.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This page intentionally blank.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX A: FUNCTION REFERENCE

This appendix provides detailed information on the primary ARINC 429 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 protocol-independent functions are prefixed by **`BTICard_`** (see Table A.1) and the ARINC 429–specific functions are prefixed by **`BTI429_`** (see Table A.2). Functions for other protocols are documented in separate manuals. The functions fall into several operational categories. The initial letters of a function name after the prefix indicate the category to which it belongs (e.g., Ch, Msg, Sched, etc.). These initial characters are followed by an action. For example, the function **`BTI429_MsgCreate`** is a member of the "Message" category and causes an ARINC 429 message to be created. In this appendix, the functions are listed alphabetically without regard to the prefix.

"handle" parameters

Nearly all functions 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`**. Please refer to Section 2.4 for a complete discussion of handles, cards, and cores.

"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., CHCFG429_DEFAULT is used in the **`BTI429_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., CHCFG429_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.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Schedule indices (SCHNDX)

All of the scheduling functions (**BTI429_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., **BTI429_SchedCall** and **BTI429_SchedBranch**).

"channel" parameters

Some functions take a *channel* parameter to specify which ARINC 429 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 429 Device or run the test program 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 structures on the Device. All of the message manipulation functions (e.g., **BTI429_MsgDataRd**) require a *message* address parameter that uniquely identifies a message structure. Several different functions (e.g., **BTI429_MsgCreate**, **BTI429_FilterSet**, **BTI429_FilterDefault**) return the message address.

The message values are 32-bit parameters specifying the value of an ARINC 429 word. The ARINC 429 word may contain a label, parity bit, SDI, SSM, and data bits. The functions expect the message value to be in a "reversed label" ARINC 429 format. The relationship between the two formats is shown below:

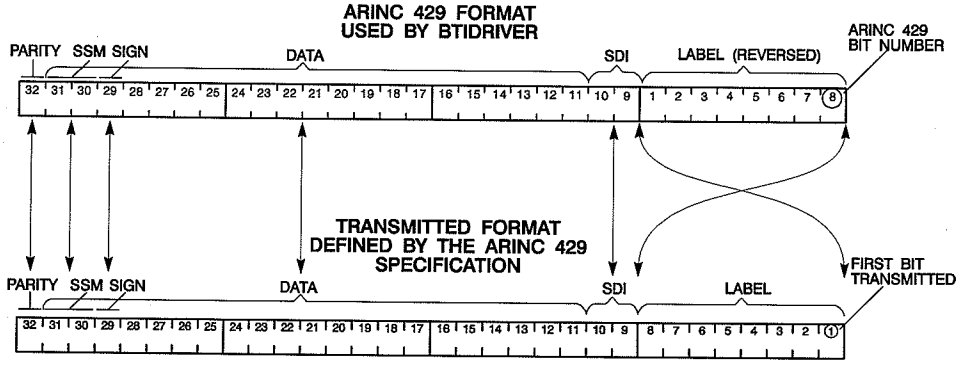


Figure A.1—Standard and reversed label formats of the ARINC 429 word

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Error values

Type ERRVAL functions return a negative value if an error occurs, or zero if successful. This value can be interpreted by BTICard_ErrDescStr, which returns a string describing the specified error. This same error information may also be obtained from the header file.

UINT16/UINT32

BTIDriver functions can be called by applications that run on both 32-bit systems and 64-bt systems, but BTIDriver functions always assume 'short' integers are 16-bits long and 'long' integers are 32 bits long. An application running on some 64-bit operating systems (for example, Linux) assumes a 'short' integer is 16-bits long, but a 'long' integer is 64-bits long. To resolve this discrepancy and to keep types consistent, applications that call BTIDriver functions from 64-bit operating systems supported by BTIDriver should use the UINT16 type instead of USHORT, and the UINT32 type instead of ULONG.

Function Summaries

Table A.1 is a summary of the **`BTICard_`** driver functions and Table A.2 summarizes the **`BTI429_`** functions. The functions that appear in bold in the following tables were illustrated in the body of this manual in the example programs.

CARD Functions	
BTICard_CardClose	Disables access to a Device and releases its hardware resources
BTICard_CardIsRunning	Checks whether core is running
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 core
BTICard_CardStart	Starts operation of the core
BTICard_CardStop	Stops operation of the core
BTICard_CardTest	Performs a hardware test on the core
BTICard_Trigger	Generates a software-simulated trigger signal on all available trigger lines
BTICard_TriggerEx	Generates a software-simulated trigger signal on the specified trigger line(s)
BTICard_CardTypeStr	Returns the type or model number of the Device
BTICard_CoreOpen	Enables access to the specified core
EVENT Functions	
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	
BTICard_ExtDIOMonConfig	Enables Sequential monitoring on specific transitions of the digital I/O pins
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
BTICard_ExtStatusLEDRd	Reads the on/off value and color of the Status LED
BTICard_ExtStatusLEDWr	Sets the on/off value and color of the Status LED
INTERRUPT Functions	
BTICard_IntClear	Clears the interrupt from the core (OS-dependent)
BTICard_IntInstall	Associates an event object with interrupts from the core (OS-dependent)
BTICard_IntUninstall	Removes association between event objects and interrupts (OS-dependent)

Table A.1— Protocol-independent BTICard_functions (continued on next page)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

IRIG TIME Functions	
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_IRIGInputThresholdGet	Gets the threshold of the IRIG input circuitry
BTICard_IRIGInputThresholdSet	Sets the threshold of the IRIG input circuitry
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_IRIGTimeBCDToBin	Converts IRIG BCD value to binary time
BTICard_IRIGTimeBinToBCD	Converts binary time value to IRIG BCD value
BTICard_IRIGWr	Sets (initializes) the IRIG timer on the specified core
SEQUENTIAL RECORD Functions	
BTICard_SeqBlkRd	Reads multiple records out of the Sequential Record (use for few records)
BTICard_SeqCommRd	Reads multiple records out of the Sequential Record (use for many records)
BTICard_SeqConfig	Configures the Sequential Monitor
BTICard_SeqDMARd	Reads multiple records out of the Sequential Record (use for many records)
BTICard_SeqFindCheckVersion	Tests the version number of the specified record
BTICard_SeqFindInit	Initializes the BTICard_SeqFindNext?? functions
BTICard_SeqFindNext	Finds the next message in the Sequential Record buffer
BTICard_SeqFindMore1553	Finds the extra 1553 record fields, when present (Device-dependent)
BTICard_SeqFindNext1553	Finds the next MIL-STD-1553 message in the Sequential Record buffer
BTICard_SeqFindNext429	Finds the next ARINC 429 message in the Sequential Record buffer
BTICard_SeqFindNext708	Finds the next ARINC 708 message in the Sequential Record buffer
BTICard_SeqFindNext717	Finds the next ARINC 717 message in the Sequential Record buffer
BTICard_SeqFindNextDIO	Finds the next DIO message in the Sequential Record buffer
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 a single record 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	
BTICard_Timer64Rd	Reads the current value of the Device timer (64 bit)
BTICard_Timer64Wr	Writes a value to the Device timer (64 bit)
BTICard_TimerClear	Clears the Device timer
BTICard_TimerRd	Reads the current value of the Device timer (32 bit)
BTICard_TimerResolution	Selects a time-tag timer resolution
BTICard_TimerStatus	Checks status of the timer configuration
BTICard_TimerWr	Writes a value to the Device timer (32 bit)
UTILITY Functions	
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.1— Protocol-independent BTICard_functions (continued from previous page)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHANNEL Functions	
BTI429_ChClear	Clears either a transmit Schedule or a receiver Filter Table
BTI429_ChConfig	Configures either a transmit or a receive channel
BTI429_ChGetCount	Gets the receiver and transmitter channel count
BTI429_ChGetInfo	Gets information about the specified channel on the specified core
BTI429_ChIs429	Checks to see if the specified channel is ARINC 429
BTI429_ChIsRcv	Checks if the specified channel is a receiver
BTI429_ChIsXmt	Checks if the specified channels is a transmitter
BTI429_ChPause	Pauses operation of a channel
BTI429_ChPauseCheck	Checks to see if the specified channel is paused
BTI429_ChResume	Resumes operation of a previously paused channel
BTI429_ChStart	Starts operation of a previously stopped channel
BTI429_ChStop	Stops operation of a channel
BTI429_ChSyncDefine	Defines the sync pulse settings for the specified channel
BTI429_ChTriggerDefine	Defines the transmit trigger settings for the specified channel
CMD Functions	
BTI429_CmdShotRd	Reads the single-shot bit for the specified index
BTI429_CmdShotWr	Sets the single-shot bit for the specified index
BTI429_CmdSkipRd	Reads the skip bit for the specified index
BTI429_CmdSkipWr	Sets the skip bit for the specified index
BTI429_CmdStepRd	Reads the step bit for the specified index
BTI429_CmdStepWr	Sets the step bit for the specified index
FILTER Functions	
BTI429_FilterDefault	Creates a default message, and points the entire table to that message
BTI429_FilterRd	Reads the message address associated with a filter location
BTI429_FilterSet	Creates a message, and points the specified filter location to that message
BTI429_FilterWr	Writes a message address to the specified filter location
LIST Functions	
BTI429_ListAsyncCreate	Creates an asynchronous transmit List Buffer
BTI429_ListDataBlkRd	Reads multiple messages from the List Buffer
BTI429_ListDataBlkWr	Writes multiple messages to the List Buffer
BTI429_ListDataRd	Reads the next data value associated with a List Buffer
BTI429_ListDataWr	Writes the next data value associated with a List Buffer
BTI429_ListRcvCreate	Creates a receive List Buffer
BTI429_ListStatus	Checks the status of the List Buffer
BTI429_ListXmtCreate	Creates a transmit List Buffer
MESSAGE Functions	
BTI429_MsgBlockRd	Reads an entire Message Record on the Device
BTI429_MsgBlockWr	Writes an entire Message Record on the Device
BTI429_MsgCommRd	Reads an entire Message Record on the Device (non-contended access)
BTI429_MsgCommWr	Writes an entire Message Record on the Device (non-contended access)
BTI429_MsgCreate	Creates and initializes a Message Record
BTI429_MsgDataRd	Reads the data associated with a message
BTI429_MsgDataWr	Writes the data associated with a message
BTI429_MsgGroupBlockRd	Reads multiple, entire Message Records in a single operation
BTI429_MsgGroupBlockWr	Writes multiple, entire Message Records in a single operation
BTI429_MsgGroupRd	Reads the data from multiple Message Records in a single operation
BTI429_MsgGroupWr	Writes the data from multiple Message Records in a single operation
BTI429_MsgIsAccessed	Returns the value of the Hit bit and then clears it
BTI429_MsgSkipRd	Reads the state of the skip bit for the specified message
BTI429_MsgSkipWr	Writes the state of the skip bit for the specified message
BTI429_MsgSyncDefine	Defines the sync pulse settings for the specified message
BTI429_MsgTriggerDefine	Defines the transmit trigger settings for the specified message
PARAMETRIC Functions	
BTI429_ParamAmplitudeConfig	Configures parametric amplitude control on the specified transmit channel
BTI429_ParamBitRateConfig	Configures parametric frequency control on the specified channel

Table A.2—ARINC 429 BTI429_ functions (continued on next page)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

SCHEDULE Functions	
BTI429_SchedBranch	Inserts a BRANCH command into the Schedule
BTI429_SchedBuild	Builds a Schedule based on minimum and maximum transmit intervals
BTI429_SchedCall	Inserts a CALL command into the Schedule
BTI429_SchedEntry	Sets the starting point of the Schedule
BTI429_SchedGap	Inserts an intermessage gap into the Schedule (calls either SchedGapFixed or SchedGapList based on the configuration of an asynchronous List Buffer)
BTI429_SchedGapFixed	Inserts a fixed gap into the Schedule.
BTI429_SchedGapList	Inserts a conditional intermessage gap into the Schedule (used for asynchronous List Buffers)
BTI429_SchedHalt	Inserts a HALT Command Block into the Schedule
BTI429_SchedLog	Inserts an LOG Command Block into the Schedule
BTI429_SchedMsg	Inserts a message into the Schedule
BTI429_SchedMsgEx	Inserts a message with a specified gap value (of less than 4 bit times)
BTI429_SchedPause	Inserts a PAUSE Command Block into the Schedule
BTI429_SchedRestart	Inserts a RESTART Command Block into the Schedule
BTI429_SchedReturn	Inserts a RETURN command into the Schedule
UTILITY Functions	
BTI429_BCDGetData	Extracts data value from BCD word
BTI429_BCDGetMant	Extracts mantissa from a BCD word
BTI429_BCDGetSign	Extracts the sign from a BCD word
BTI429_BCDGetSSM	Extracts SSM field from BCD word
BTI429_BCDGetVal	Calculates the value of a BCD word
BTI429_BCDPutData	Inserts data value into BCD word
BTI429_BCDPutMant	Inserts mantissa value into BCD word
BTI429_BCDPutSign	Inserts the sign into a BCD word
BTI429_BCDPutSSM	Inserts SSM field into BCD word
BTI429_BCDPutVal	Inserts the value into a BCD word
BTI429_BNRGetData	Extracts data value from BNR word
BTI429_BNRGetMant	Extracts mantissa from a BNR word
BTI429_BNRGetSign	Extracts the sign from a BNR word
BTI429_BNRGetSSM	Extracts SSM field from BNR word
BTI429_BNRGetVal	Calculates the value of a BNR word
BTI429_BNRPutData	Inserts data value into BNR word
BTI429_BNRPutMant	Inserts mantissa value into BNR word
BTI429_BNRPutSign	Inserts the sign into a BNR word
BTI429_BNRPutSSM	Inserts the SSM field into a BNR word
BTI429_BNRPutVal	Inserts the value into a BNR word
BTI429_FldGetData	Extracts data field from ARINC 429 word
BTI429_FldGetLabel	Extracts label field from ARINC 429 word
BTI429_FldGetParity	Extracts parity bit from ARINC 429 word
BTI429_FldGetSDI	Extracts SDI field from ARINC 429 word
BTI429_FldGetValue	Extracts specified field from ARINC 429 word
BTI429_FldPutData	Inserts the data field into an ARINC 429 word
BTI429_FldPutLabel	Inserts the label field into an ARINC 429 word
BTI429_FldPutSDI	Inserts the SDI field into an ARINC 429 word
BTI429_FldPutValue	Inserts the specified field into an ARINC 429 word

Table A.2—ARINC 429 BTI429_ functions (continued from previous page)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Device Dependence

The BTIDriver unified API supports many generations of Ballard hardware Devices. This cross-compatibility allows for application reuse when migrating from one Device to another. Each successive generation of Ballard hardware Devices tries to build upon the feature set of the previous one. Therefore, not all features supported by this API apply to all hardware Devices. Functions that depend upon a particular hardware Device will reference the products listed in Table A.3 by generation or by other functionality.

	Generation/Group			
Product	**3G**	**4G**	**5G**	**RPC**
BUSBox (BB1xxx)	✓			
OmniBus PCI (111-xxx, 112-xxx-xxx)		✓		
OmniBus cPCI (121-xxx, 122-xxx-xxx)		✓		
OmniBus PMC (141-xxx)		✓		
OmniBus VME (152-xxx-xxx, 154-xxx-xxx-xxx-xxxx)		✓		✓
OmniBusBox (162-xxx-xxx)		✓		✓
Avionics BusBox 1000 (AB1xxx)			✓	✓
Avionics BusBox 2000 (AB2xxx)			✓	✓
Lx1553-5, Lx429-5, PM429-2			✓	
USB 1553, USB 429/717 (UA1xxx)			✓	

Table A.3—Devices grouped by generation and functionality

Function Detail

The following pages contain descriptions of the BTIDriver functions (in alphabetical order without regard to prefix). The constants in bold in the tables are the default options. Note that the "**`BTICard_`**" and "**`BTI429_`**" prefixes have been omitted from the headings for easier reading, but all BTIDriver functions must begin with the appropriate prefix in source code.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDGetData

```
ULONG BTI429_BCDGetData(
   ULONG msg,                    //BCD word to extract data from
   USHORT msb,                   //Most significant bit of BCD field
   USHORT lsb                    //Least significant bit of BCD field
)
```

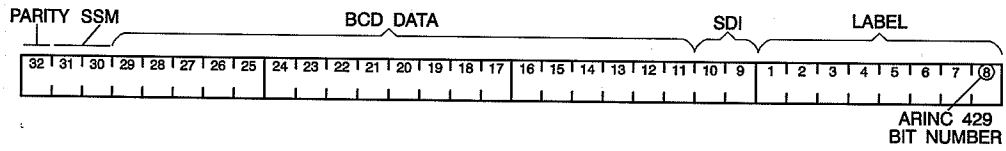
RETURNS

32-bit value of data field containing up to a maximum of 21 bits (19 bits from the BCD data field plus the 2-bit SDI field).

DESCRIPTION

Extracts the data field of the BCD word in *msg*. *msb* and *lsb* specify the ARINC 429 bit numbers of the most significant and least significant bits of the BCD field respectively. The specified bits are converted to a 32-bit unsigned value. No other conversions, such as resolution or sign, are made.

The function assumes the BCD word has the following format:



The bits from the parity and SSM fields are not be included in the return value. When specifying a *lsb* value in the SDI field (bits 9 or 10), the bits are treated as data for this function.

Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

lsb and *msb* values are limited to the BCD data/SDI field (bits 9 through 29).

SEE ALSO

BTI429_BCDPutData, BTI429_BCDGetSign

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDGetMant

```
ULONG BTI429_BCDGetMant(
   ULONG msg,                    //BCD word to extract data from
   USHORT sigdig                 //Number of significant digits
)
```

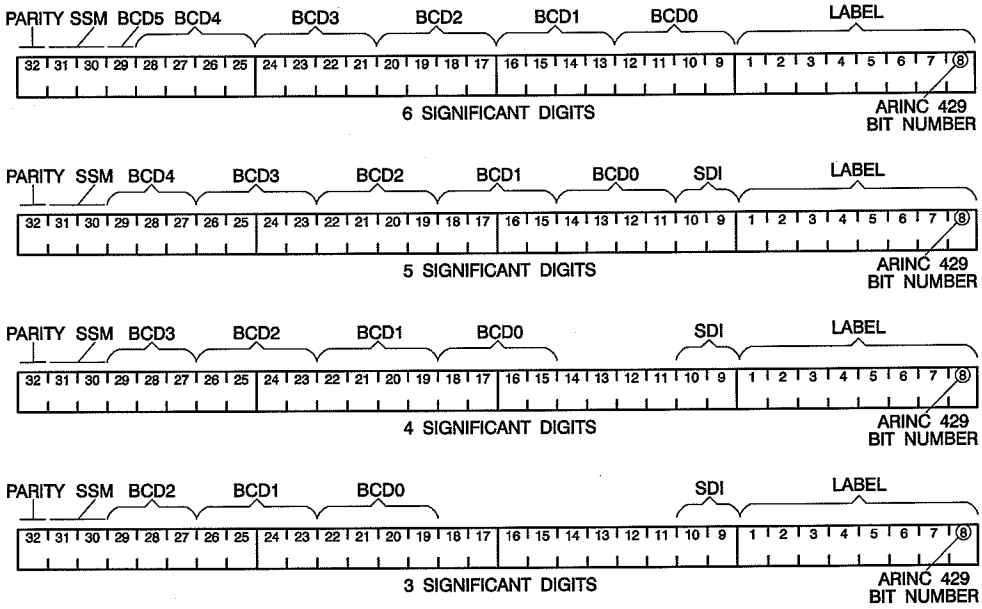
RETURNS

The data field mantissa.

DESCRIPTION

Extracts the mantissa value from the data field of the BCD word specified in *msg*. *sigdig* specifies the number of significant BCD digits (characters) in the data field. The result is converted to a 32-bit unsigned value and returned. No other conversion, such as scaling by the resolution value, is made.

The function assumes the BCD data field is divided into the following fields:



Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Per the ARINC 429 specification, the *sigdig* value is defined for the following number of significant digits: 3, 4, 5, and 6. Use the BTI429_BCDGetData function for other non-standard data formats.

SEE ALSO

BTI429_BCDPutMant

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDGetSign

```
USHORT BTI429_BCDGetSign(
   ULONG msg                    //BCD word to extract data from
)
```

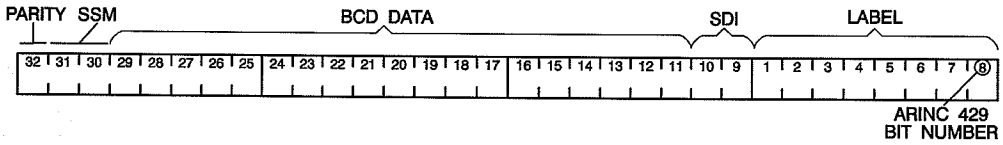
RETURNS

A non-zero value if sign of BCD word is negative, otherwise zero if the sign is positive.

DESCRIPTION

Extracts the sign of the BCD word specified in *msg*. The result is non-zero if the sign of the BCD word is negative (SSM field equals 11 binary). Otherwise, the function returns a zero value.

The function assumes the SSM field is located at bits 30 through 31 as shown below:



Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_BCDPutSign

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDGetSSM

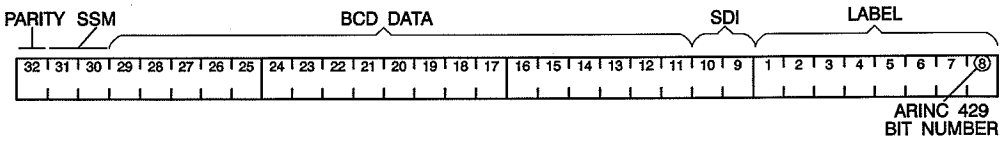
```
USHORT BTI429_BCDGetSSM(
   ULONG msg                    //BCD word to extract data from
)
```

RETURNS

Value of the 2-bit SSM field.

DESCRIPTION

Extracts the SSM field of the BCD word in *msg*. The function assumes the SSM field is located at bits 30 through 31 as shown below:



Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_BCDPutSSM

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDGetVal

```
VOID BTI429_BCDGetVal(
   LPSTR buf,              //Pointer to buffer to receive ASCII string
   ULONG msg,              //BCD word to extract data from
   USHORT sigdig,          //Number of significant digits
   LPCSTR resolstr         //Pointer to resolution string
)
```

RETURNS

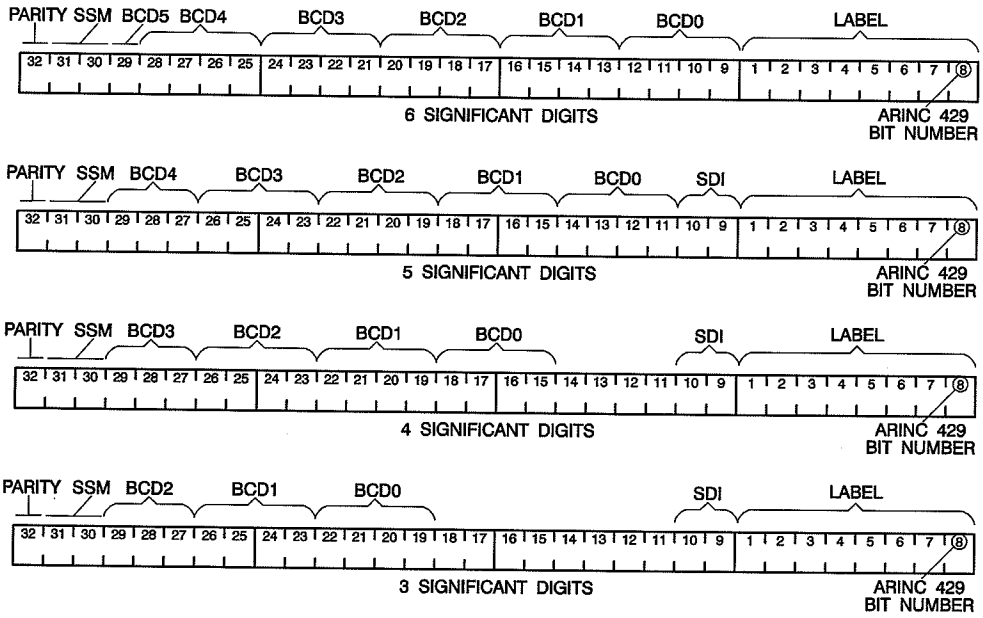
None.

DESCRIPTION

Extracts the data field of the BCD word specified in *msg* and calculates the value. The resulting ASCII string is written to *buf*. This string may contain a decimal point and may be signed.

sigdig specifies the number of significant digits (characters) in the BCD data field. *resolstr* points to a null-terminated ASCII string specifying the resolution of the BCD data. This resolution string is a decimal real number that represents the value of the least significant digit (e.g., 0.1). It may contain a decimal point if needed, but should not have a sign or exponent.

The function assumes the BCD data field is divided into the following fields:



Note: This is a utility function and does not access any Device hardware.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Per the ARINC 429 specification, the *sigdig* value is defined for the following number of significant digits: 3, 4, 5, and 6. Use of functions BTI429_BCDGetData and BTI429_BCDGetSign and additional scaling by the resolution value may be required for non-standard data formats.

SEE ALSO

BTI429_BCDPutVal, BTI429_BCDGetData, BTI429_BCDGetSign

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDPutData

```
ULONG BTI429_BCDPutData(
   ULONG msg,                  //32-bit BCD word
   ULONG value,                //New data value
   USHORT msb,                 //Most significant bit of BCD field
   USHORT lsb                  //Least significant bit of BCD field
)
```

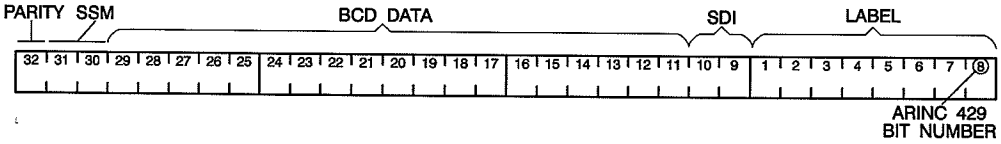
RETURNS

The new 32-bit BCD word with the data field inserted.

DESCRIPTION

Inserts *value* into the data field of the BCD word specified by *msg*. *msb* and *lsb* specify the ARINC 429 bit numbers of the most significant and least significant bits of the field respectively. *value* is converted to BCD and inserted into the specified bits (within bits 9 through 29). No other conversion is made.

The function assumes the BCD word has the following format:



Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

lsb and *msb* values are limited to the BCD data/SDI field (bits 9 through 29). A maximum of 21 bits (when *msb*=29 and *lsb*=9) of *value* is used by this function.

SEE ALSO

BTI429_BCDGetData

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDPutMant

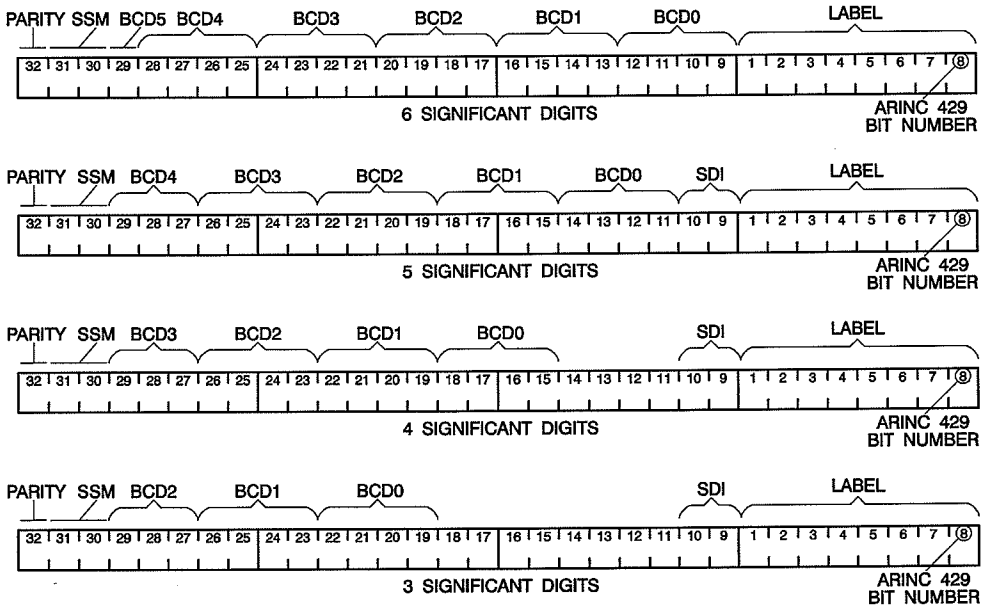
```
ULONG BTI429_BCDPutMant(
   ULONG msg,                    //32-bit BCD word
   ULONG value,                  //New data value
   USHORT sigdig,                //Number of significant digits
   USHORT sign                   //Sign for SSM field
)
```

RETURNS

The new 32-bit BCD word with the data field inserted.

DESCRIPTION

Inserts the mantissa value into the data field of the BCD word specified in *msg*. *sigdig* specifies the number of significant digits (characters) in the BCD field. *value* is converted to BCD and inserted into the data field. The BCD data field is assumed to be divided into the following fields:



If sign is non-zero, the value 11 (binary) is inserted into the SSM field to denote a signed value. Otherwise, the value 00 (binary) is inserted into the SSM field. No other conversion is made.

Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_BCDGetMant

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDPutSign

```
ULONG BTI429_BCDPutSign(
   ULONG msg,                    //32-bit BCD word
   USHORT sign                   //Sign for SSM field
)
```

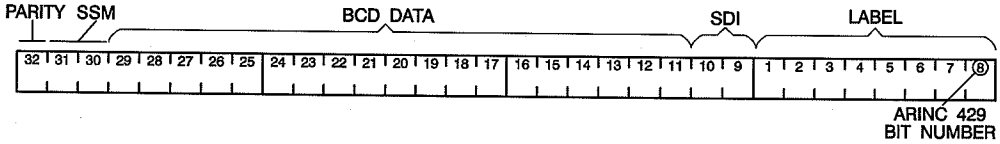
RETURNS

The new 32-bit BCD word with the SSM field inserted.

DESCRIPTION

Inserts *sign* into the SSM field of the BCD word specified in *msg*. If *sign* is non-zero, the value 11 (binary) is inserted into the SSM field to specify a signed value. Otherwise, the value 00 (binary) is inserted into the SSM field.

The function assumes the SSM field is located at bits 30 through 31 as shown below:



Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_BCDGetSign

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDPutSSM

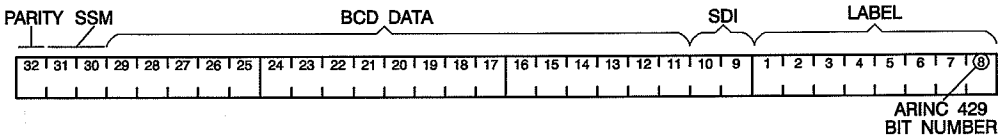
```
ULONG BTI429_BCDPutSSM(
   ULONG msg,                    //32-bit BCD word
   USHORT value                  //2-bit value of SSM field
)
```

RETURNS

The new 32-bit BCD word with the SSM field inserted.

DESCRIPTION

Inserts value into the SSM field of the BCD word in msg. The function assumes the SSM field is located at bits 30 through 31 as shown below:



Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_BCDGetSSM

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BCDPutVal

```
ULONG BTI429_BCDPutVal(
   LPCSTR buf,              //Pointer to buffer containing ASCII string
   ULONG msg,               //32-bit BCD word
   USHORT sigdig,           //Number of significant digits
   LPCSTR resolstr          //Pointer to resolution string
)
```

RETURNS

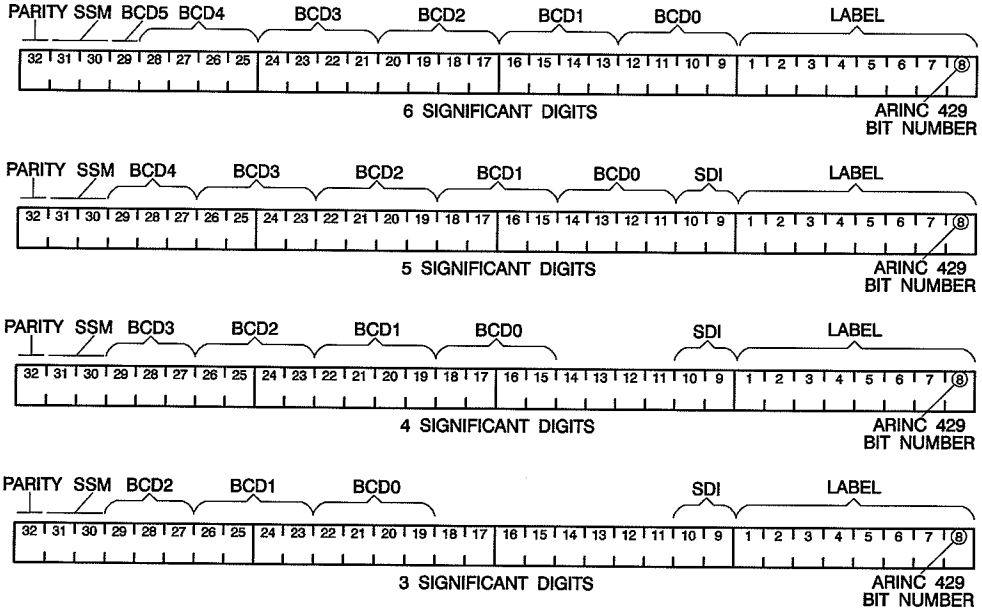
The new 32-bit BCD word with the data field inserted.

DESCRIPTION

Inserts a new value into the data field of the BCD word specified by *msg*. *buf* holds an ASCII string that contains the value to insert. It may contain a decimal point and may be signed.

sigdig specifies the number of significant digits (characters) in the BCD data field. *resolstr* points to a null-terminated ASCII string specifying the resolution of the BCD data. This resolution string is a decimal real number that represents the value of the least significant digit (e.g., 0.1). It may contain a decimal point if needed, but should not have a sign or exponent.

The BCD data field is assumed to be divided into the following fields:



Note: This is a utility function and does not access any Device hardware.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Per the ARINC 429 specification, the *sigdig* value is defined for the following number of significant digits: 3, 4, 5, and 6. Use of functions BTI429_BCDPut-Data and BTI429_BCDPutSign and additional scaling by the resolution value may be required for non-standard data formats.

SEE ALSO

BTI429_BCDGetVal, BTI429_BCDPutData, BTI429_BCDPutSign

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BNRGetData

```
ULONG BTI429_BNRGetData(
   ULONG msg,                    //BNR word to extract data from
   USHORT msb,                   //Most significant bit of BNR field
   USHORT lsb                    //Least significant bit of BNR field
)
```

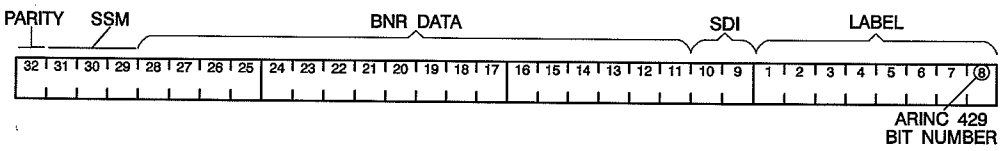
RETURNS

32-bit value of data field containing up to a maximum of 20 bits (18 bits from the BCD data field plus the 2-bit SDI field).

DESCRIPTION

Extracts the data field of the BNR word in *msg*. *msb* and *lsb* specify the ARINC 429 bit numbers of the most significant and least significant bits of the BNR data field respectively. The specified bits are converted to a 32-bit unsigned value. No other conversions, such as resolution or sign, are made.

The function assumes the BNR word has the following format:



The bits from the parity and SSM fields are not be included in the return value. When specifying a *lsb* value in the SDI field (bits 9 or 10), the bits are treated as data for this function.

Note: This is a utility function and does not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

lsb and *msb* values are limited to the BNR data/SDI field (bits 9 through 28).

SEE ALSO

BTI429_BNRPutData, BTI429_BNRGetSign

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP