ARINC 429
PROGRAMMING MANUAL

for BTIDriver-Compliant Devices

November 21, 2013
Rev. E

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*

MA132-20131121 Rev. E

COPYRIGHT NOTICE

SAFETY WARNING

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

TRADEMARKS

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

This page intentionally blank.

1. INTRODUCTION

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.

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.

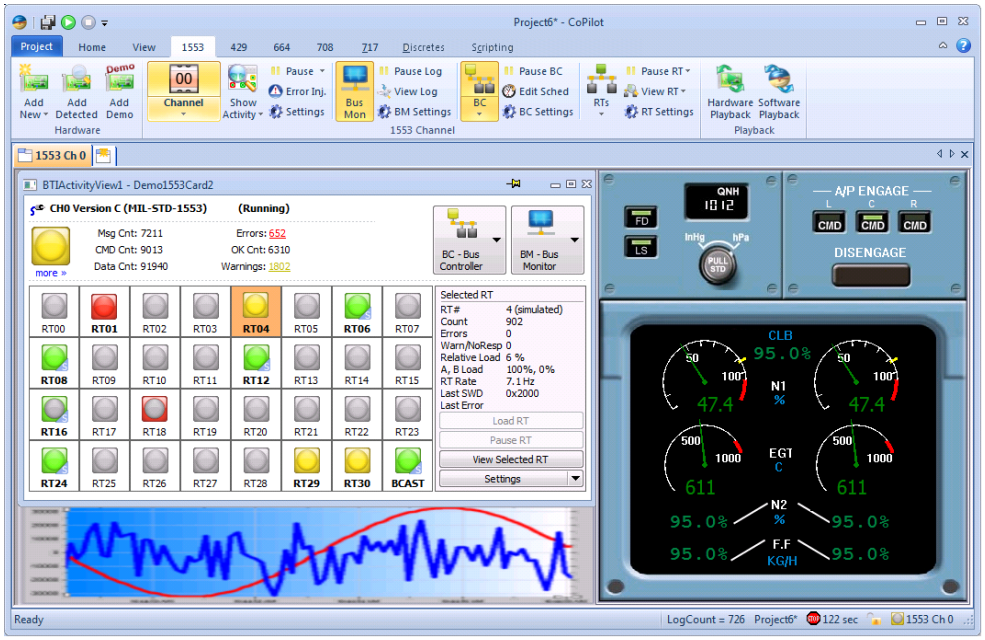


Figure 1.1—A sample CoPilot screen

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 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.

This page intentionally blank.

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."

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.

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.

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**.

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.

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 **BTI-429_FilterSet** functions assign receive messages to Message Records by placing entries in the Filter Table. The **BTI429_FilterDefault** and **BTI-429_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 **BTI-Card_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.

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

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 **BTI-429_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 **BTI-Card_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.

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`**.

```
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);
    }

    //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.

This page intentionally blank.

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.

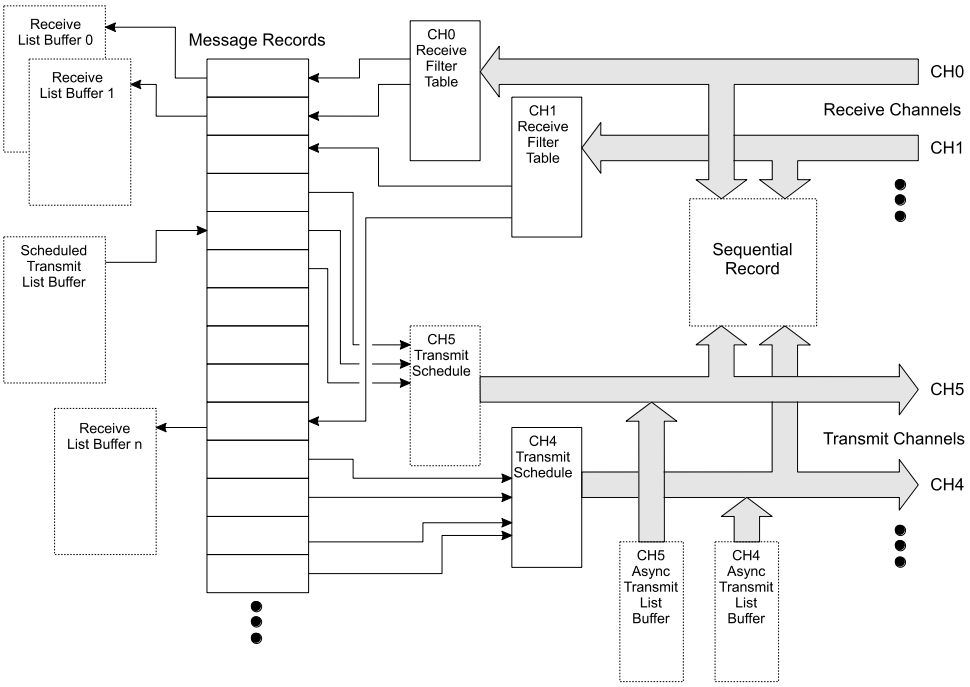


Figure 3.1—Message flow between primary data structures in a typical 429 Device

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.

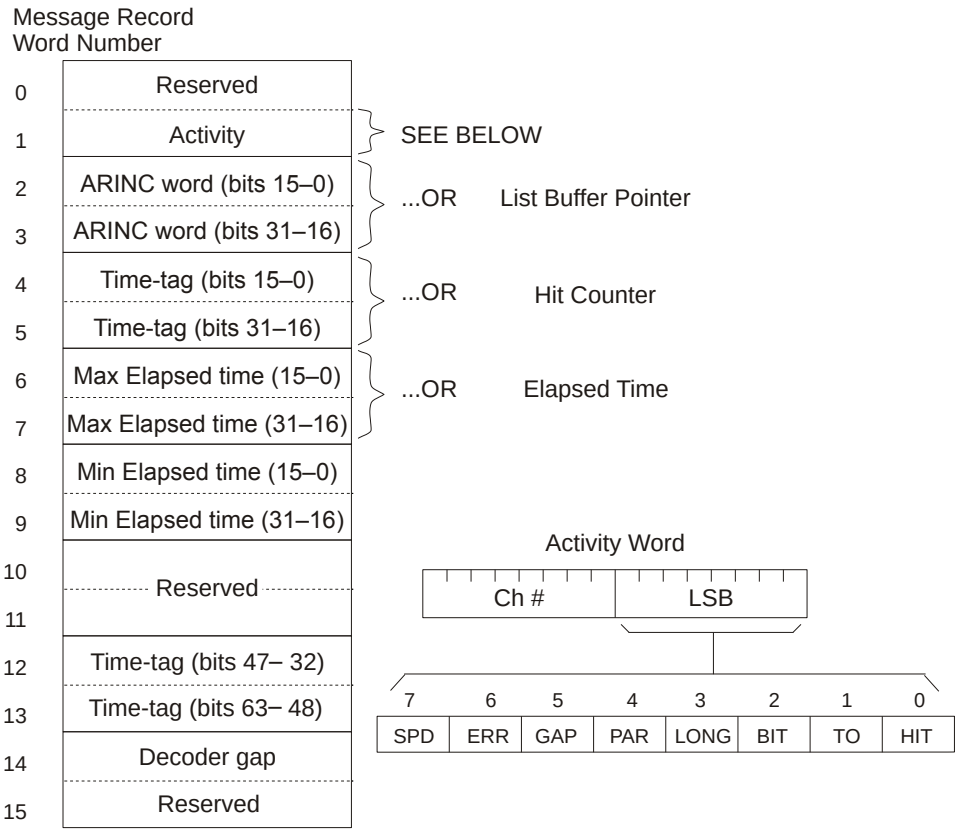


Figure 3.2—Message Record structure

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.

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.

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 en-

abled. For example, if the Elapsed Time option is enabled, the Device calculates the time elapsed since the last reception of the word and writes this to the Message Record.

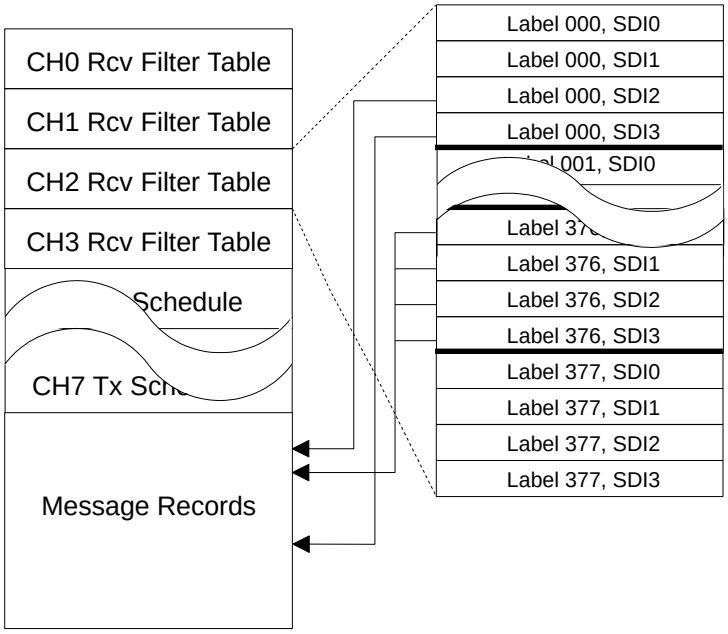


Figure 3.3—Filter Tables

3.3.2 Configuring the Filter Tables

The user sets Filter Table entries to point to Message Records with the **`BTI429_FilterSet`** and **`BTI429_FilterDefault`** functions. **`BTI429_ChConfig`** 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.

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.

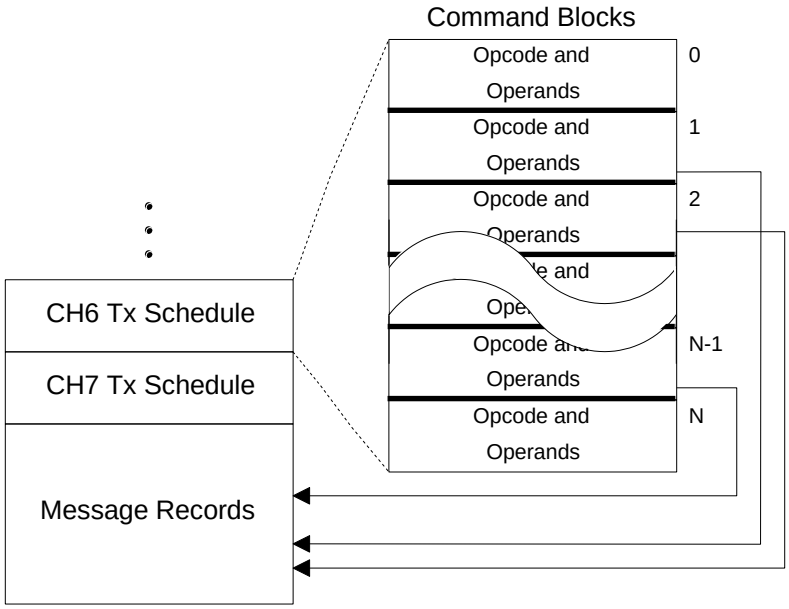


Figure 3.4—Transmit Schedule

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).

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`**, **`BTI429_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

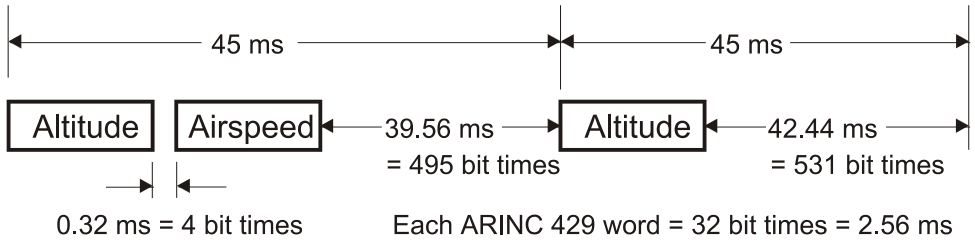


Figure 3.5—Transmit Schedule for the low-speed timing requirements of Table 3.2

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.

```
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-

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-**

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.

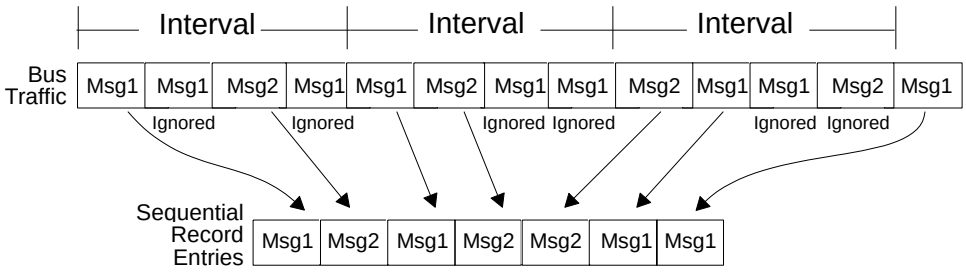


Figure 3.7—Operation of Sequential Record in Interval mode

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.

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**.

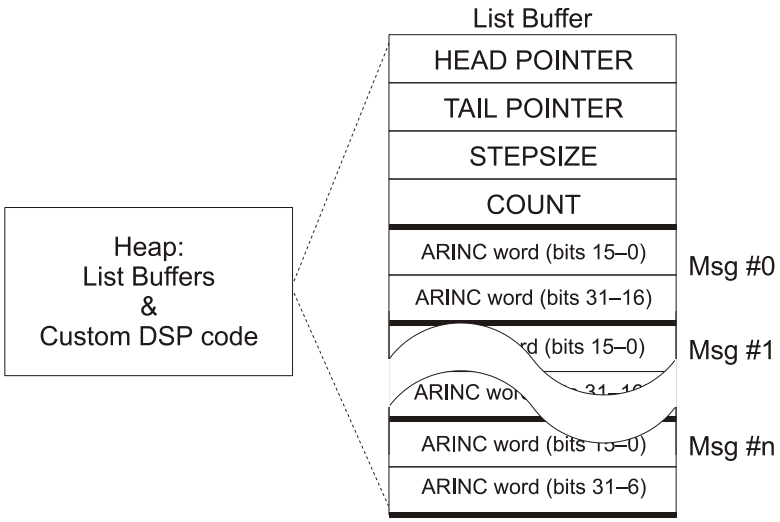


Figure 3.8—Receive and transmit List Buffer structure

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.

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 **BTI429_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.

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.

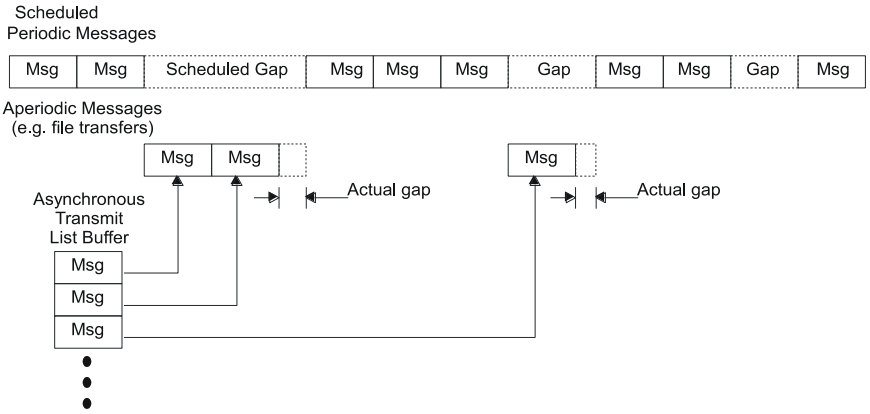


Figure 3.9—Operation of an asynchronous transmit List Buffer

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.

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.

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.

This page intentionally blank.

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 bit-mask 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.

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:

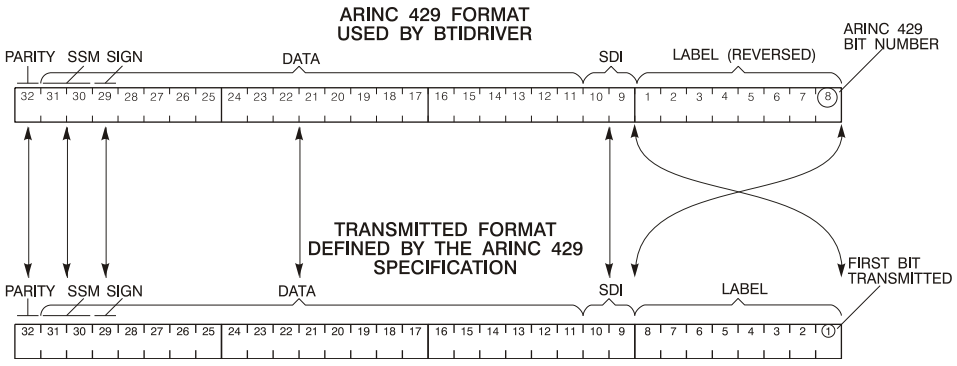


Figure A.1—Standard and reversed label formats of the ARINC 429 word

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.

BIT Functions	
BTICard_BITConfig	Configures the Built-In Test functionality of the card
BTICard_BITInitiate	Performs a hardware test on the specified card
BTICard_BITStatusClr	Clears the historical maximum and minimum values
BTICard_BITStatusRd	Reads the current status of Built-In Test (BIT)
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_CardTrigger	Generates a software-simulated trigger signal on all available trigger lines
BTICard_CardTriggerEx	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

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

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)
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
SYSMON Functions	
BTICard_SysMonDescGet	Returns the description of a sensor
BTICard_SysMonMaxRd	Reads the historical maximum value of a sensor
BTICard_SysMonMinRd	Reads the historical minimum value of a sensor
BTICard_SysMonNomRd	Reads the nominal value of a voltage sensor
BTICard_SysMonThresholdGet	Reads the user thresholds of a temperature sensor
BTICard_SysMonThresholdSet	Sets the user thresholds of a temperature sensor
BTICard_SysMonTypeGet	Returns the type of sensor
BTICard_SysMonUserStr	Returns a formatted value string of sensor data
BTICard_SysMonValRd	Reads the current sensor value
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)

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_MsgMultiSkipWr	Writes the state of the skip bit for multiple messages
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)

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_SchedMode	Sets options for the SchedBuild function
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_SchedPulse	Inserts a discrete PULSE 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)

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)			✓	✓
AB3000 Series (AB3xxx)			✓	✓
Lx1553-5, Lx429-5, PM1553-5, PM429-2			✓	
USB 1553, USB 429/717, USB 708, USB Multi (UA1xxx)			✓	
Mx5 (Mx5x-xx-xx)			✓	

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.

BITConfig

```
ERRVAL BTICard_BITConfig(
   ULONG ctrlflags,          //Selects configuration options
   HCARD hCard               //Card handle
)
```

RETURNS

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

DESCRIPTION

Configures the Continuous Built-In Test (CBIT) functionality as defined by *ctrlflags* (see table below) for the card specified by *hCard*. Various functional blocks in the CBIT system can be enabled or disabled using the flags in the table below. Each of these blocks report to the BIT Status register that can be read using BTICard_BITStatusRd. These blocks can also be configured to generate an Event Log list entry when an error occurs.

ctrlflags	
Constant	**Description**
BITCFG_DEFAULT	Select all default settings (**bold** below)
BITCFG_MEMECC_ENABLE	Enables the memory interface to operate in ECC mode. In this mode, if a single bit error occurs, the read value will be corrected and BITSTAT_SINGLE_BIT_ERR will be set in the BIT Status register. If a double bit error occurs, the read value can't be corrected so the card will be stopped and BITSTAT_DOUBLE_BIT_ERR will be set in the BIT Status register.
BITCFG_MEMECC_DISABLE	Disables ECC operation of the memory.
BITCFG_FPGA_ENABLE	Enables monitoring for Single Event Upsets (SEU) in the FPGA configuration. In this mode, if a Single Event Upset is detected the card will be stopped and BITSTAT_CBIT_FPGA_ERR will be set in the BIT Status register.
BITCFG_FPGA_DISABLE	Disables monitoring for Single Event Upsets in the FPGA configuration.
BITCFG_PROTOCOL_ENABLE	Enables CBIT in the 1553 Protocol Engine. Every 1553 word transmitted by the Card will be monitored and checked for accuracy. If the transceiver is damaged or there is a collision on the bus, the protocol error bit will be set in the the BIT Status register. In addition, the MSGERR1553_SYSTEM bit will also be set in the 1553 Message Record and Sequential Record of the transmission that failed.
BITCFG_PROTOCOL_DISABLE	Disables CBIT in the 1553 Protocol Engine.
BITCFG_MEMECC_NOLOG	Does not generate an event log entry when the ECC decoder detects a single or double bit error in the on-card memory.
BITCFG_MEMECC_LOG	Generates an event log entry when the ECC decoder detects a single or double bit error in the on-card memory.
BITCFG_FPGA_NOLOG	Does not generate an event log entry when an SEU is detected in the FPGA Configuration.
BITCFG_FPGA_LOG	Generates an event log entry when an SEU is detected in the FPGA Configuration.
BITCFG_PROTOCOL_NOLOG	Does not generate an event log entry when 1553 CBIT detects an error.
BITCFG_PROTOCOL_LOG	Generates an event log entry when 1553 CBIT detects an error.

(Continued on next page)

(Continued from previous page)

BITCFG_CARD_STOPPED_NOLOG	Does not generate an event log entry when the Card is stopped due to CBIT Errors.
BITCFG_CARD_STOPPED_LOG	Generates an event log entry when the Card is stopped due to CBIT Errors. When a temperature or voltage sensor value exceeds the System limits for safe operation, the Card will automatically stop protocol activity to reduce power draw. The Card will also stop when an uncorrectable (double bit) error is detected in the memory or an SEU is detected in the FPGA configuration. The card is stopped to prevent the transmission of corrupted data.
BITCFG_SYSMON_NOLOG	Does not generate an event log entry when SysMon detects that a temperature sensor has exceeded the user definable thresholds.
BITCFG_SYSMON_LOG	Generates an event log entry when the SysMon detects that a temperature sensor has exceeded the user definable thresholds. User definable thresholds can be configured by calling BTICard_SysMonThresholdSet.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

This function will clear any errors in the status read by BTICard_BITStatusRd. In order to generate Event Log list entries, the Event Log must be configured by calling BTICard_EventLogConfig.

SEE ALSO

BTICard_BITStatusRd, BTICard_EventLogConfig, BTICard_SysMonThresholdSet

BITInitiate

```
ERRVAL BTICard_BITInitiate(
   HCARD hCard                    //Card handle
)
```

RETURNS

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

DESCRIPTION

Executes a read and write memory test on the card specified by *hCard*. When the test completes, the Card is left in the same state as after a call to BTICard_-CardReset.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

Do not call when the Card is connected to an active databus. The function disrupts normal databus operation of the Card, and the results will be unpredictable.

SEE ALSO

BTICard_CardReset

BITStatusClear

```
ERRVAL BTICard_BITStatusClear(
   ULONG statval,              //Mask of bits to clear
   HCARD hCard                 //Card handle
)
```

RETURNS

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

DESCRIPTION

Each bit in *statval* clears the corresponding bit in the BIT Status register for the Card specified by *hCard*. The constants in the table below may be bitwise OR-ed together to clear each specified status bit.

Constant	Description
BITSTAT_CARD_STOPPED	CBIT fatal system error, card stopped
BITSTAT_CBIT_FPGA_ERR	CBIT FPGA Single Event Upset (SEU)
BITSTAT_CBIT_PROTOCOL_ERR	CBIT Protocol Error
BITSTAT_SINGLE_BIT_ERR	CBIT ECC Single Bit Error (correctable)
BITSTAT_DOUBLE_BIT_ERR	CBIT ECC Double Bit Error
BITSTAT_SYSMON_ERR	CBIT SysMon Error

Please refer to BTICard_BITStatusRd for more information on these error flags.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_BITStatusRd

BITStatusRd

```
ULONG BTICard_BITStatusRd(
   HCARD hCard                  //Card handle
)
```

RETURNS

The value of the BIT Status register.

DESCRIPTION

Reads the BIT Status register of the Card specified by *hCard*. The status value can be tested using the following predefined constants:

Constant	Description
BITSTAT_CARD_STOPPED	PBIT/CBIT fatal system error, card stopped
BITSTAT_CBIT_FPGA_ERR	CBIT FPGA Single Event Upset (SEU)
BITSTAT_CBIT_PROTOCOL_ERR	CBIT Protocol Error
BITSTAT_SINGLE_BIT_ERR	PBIT/CBIT ECC Single Bit Error (corrected)
BITSTAT_DOUBLE_BIT_ERR	PBIT/CBIT ECC Double Bit Error
BITSTAT_SYSMON_ERR	CBIT SysMon Error

For SysMon errors, the position of the sensor's error bit is determined by the sensor index number.

```
if (BTICard_BITStatusRd(hCard) & (1 << index))
{
     // Handle error on the sensor at <index>
}
```

For more information on the other BIT errors and enabling/disabling reporting refer to BTICard_BITConfig.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_BITStatusClear, BTICard_BITConfig

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
)
```

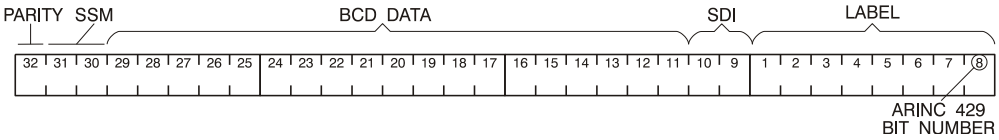
RETURNS

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

BCDGetMant

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

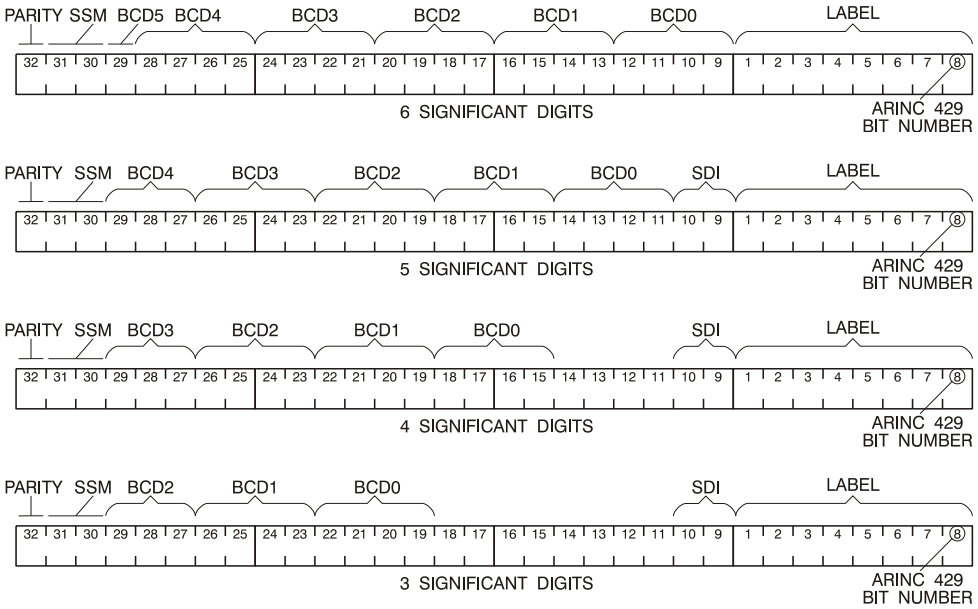
RETURNS

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

BCDGetSign

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

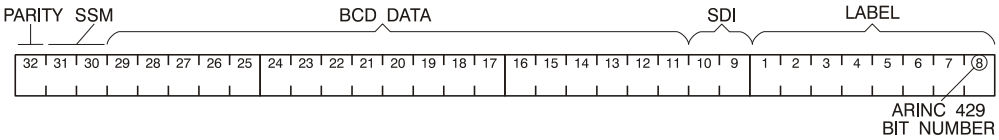
RETURNS

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

BCDGetSSM

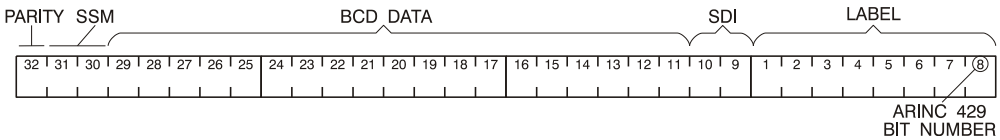
```
USHORT BTI429_BCDGetSSM(
   ULONG msg                    //BCD word to extract data from
)
```

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

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

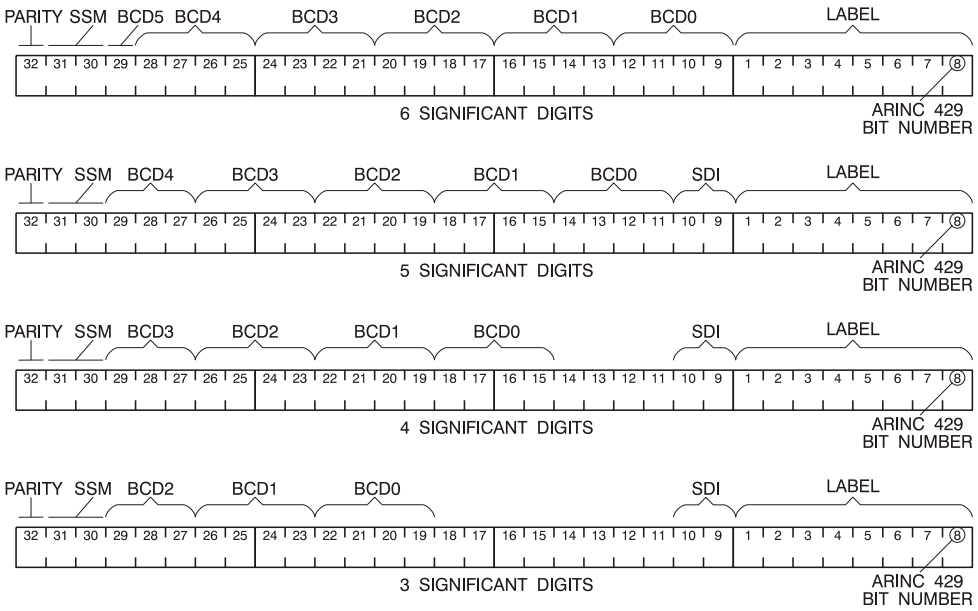
None.

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.

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

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
)
```

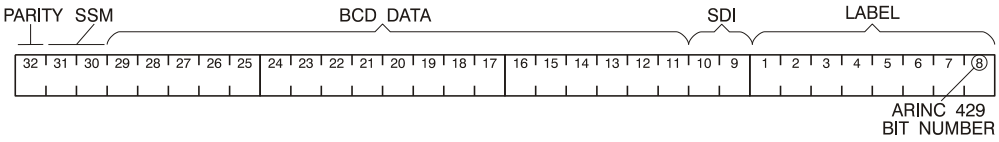
RETURNS

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

BCDPutMant

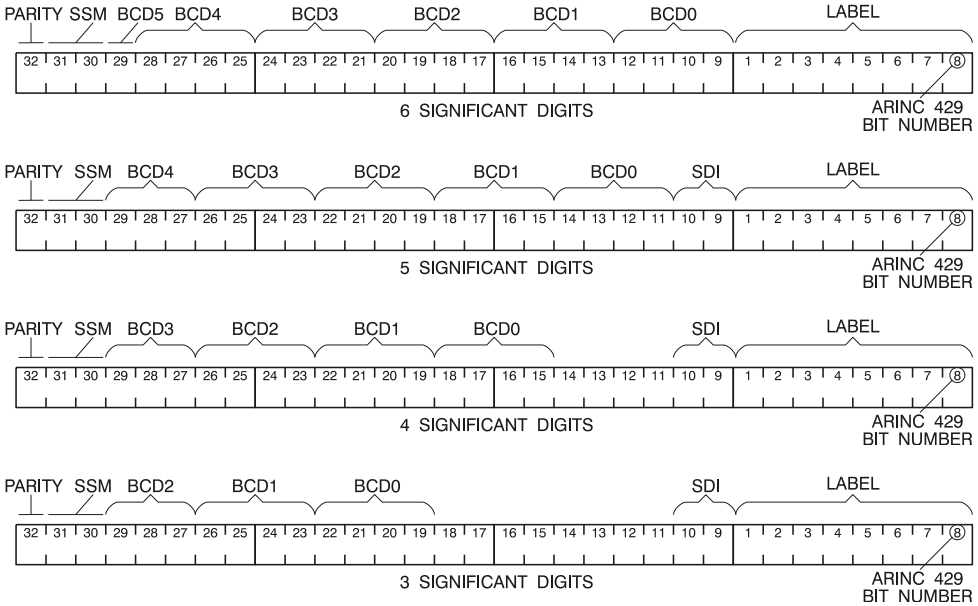
```
ULONG BTI429_BCDPutMant(
   ULONG msg,                  //32-bit BCD word
   ULONG value,                //New data value
   USHORT sigdig,              //Number of significant digits
   USHORT sign                 //Sign for SSM field
)
```

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

BCDPutSign

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

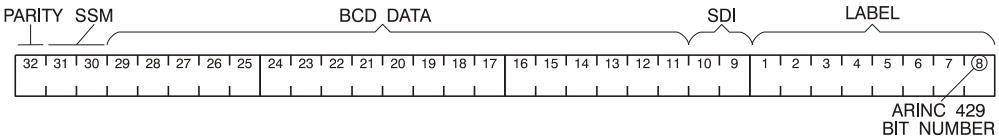
RETURNS

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

BCDPutSSM

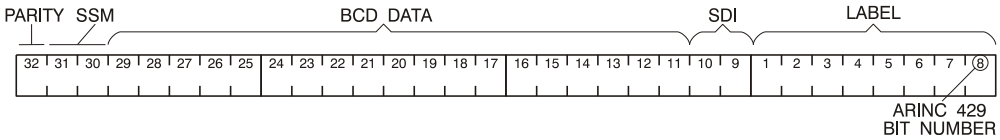
```
ULONG BTI429_BCDPutSSM(
   ULONG msg,                    //32-bit BCD word
   USHORT value                  //2-bit value of SSM field
)
```

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

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

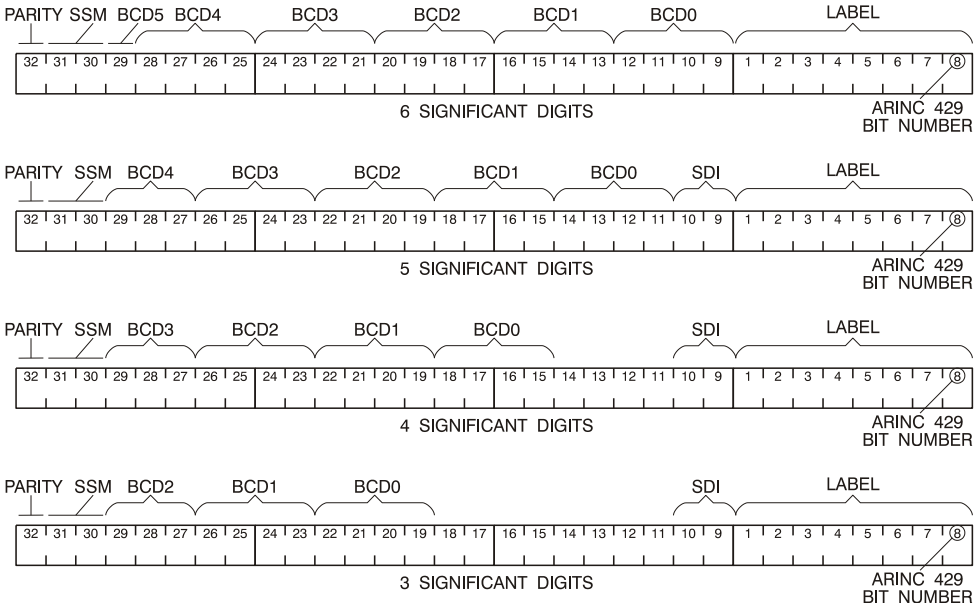
The new 32-bit BCD word with the data field inserted.

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.

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

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
)
```

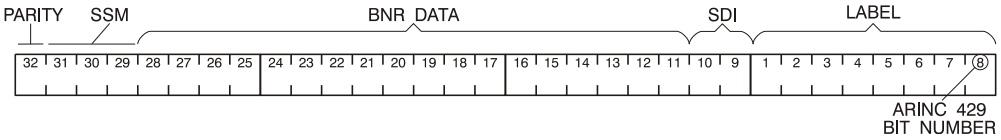
RETURNS

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

BNRGetMant

```
ULONG BTI429_BNRGetMant(
   ULONG msg,                    //BNR word to extract data from
   USHORT sigdig                 //Number of significant digits
)
```

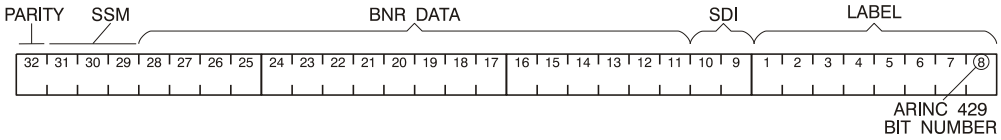
RETURNS

The data field mantissa (right-adjusted).

DESCRIPTION

Extracts the mantissa value from the data field of the BNR word specified in *msg*. *sigdig* specifies the number of significant digits in the data field. If the SSM field of *msg* specifies a signed value, then the two's complement of the data field is returned. No other conversion, such as scaling by the resolution value, is made.

The BNR data field is signed if bit 29 in the SSM field is non-zero. The BNR data field is assumed to be left-adjusted at bit 28 as shown below:



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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

The *sigdig* values may range from 1 to 20 bits.

SEE ALSO

BTI429_BNRPutMant

BNRGetSign

```
USHORT BTI429_BNRGetSign(
   ULONG msg                    //BNR word to extract data from
)
```

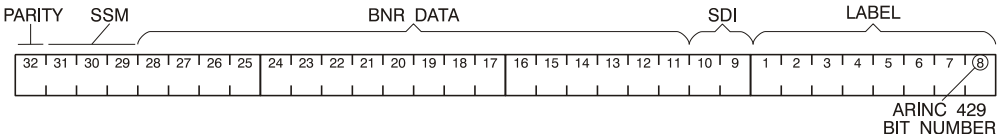
RETURNS

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

DESCRIPTION

Extracts the sign of the BNR word specified in *msg*. The result is non-zero if the sign of the BNR word is negative (bit 29 of the SSM field is non-zero). Otherwise, the function returns a zero value.

The function assumes the SSM field is located at bits 29 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_BNRPutSign, BTI429_BNRGetSSM

BNRGetSSM

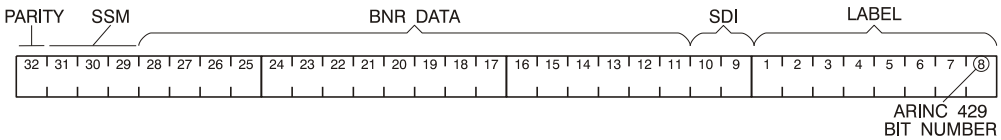
```
USHORT BTI429_BNRGetSSM(
   ULONG msg                    //BNR word to extract SSM field from
)
```

RETURNS

Value of SSM field.

DESCRIPTION

Extracts the SSM field from the BNR word in msg. The function assumes the SSM field is located at bits 29 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_BNRPutSSM, BTI429_BNRGetSign

BNRGetVal

```
VOID BTI429_BNRGetVal(
   LPSTR buf,                  //Pointer to buffer to receive ASCII string
   ULONG msg,                  //BNR word to extract data from
   USHORT sigbit,              //Number of significant bits
   LPCSTR resolstr             //Pointer to resolution string
)
```

RETURNS

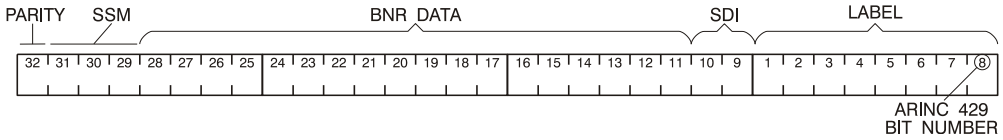
None.

DESCRIPTION

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

sigbit specifies the number of significant bits in the BNR data field. *resolstr* points to a null-terminated ASCII string specifying the resolution of the BNR data. This resolution string is a decimal real number that represents the value of the least significant bit (e.g., 0.25). It may contain a decimal point if needed, but should not have a sign or exponent.

The function assumes the data field is left-adjusted at bit 28 as shown below:



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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

The *sigdig* values may range from 1 to 20 bits. The sign of the result is determined by the SSM fields as described by the BTI429_BNRGetSign function.

SEE ALSO

BTI429_BNRPutVal, BTI429_BNRGetSign

BNRPutData

```
ULONG BTI429_BNRPutData(
   ULONG msg,                  //32-bit BNR word
   ULONG value,                //New data value
   USHORT msb,                 //Most significant bit of BNR field
   USHORT lsb                  //Least significant bit of BNR field
)
```

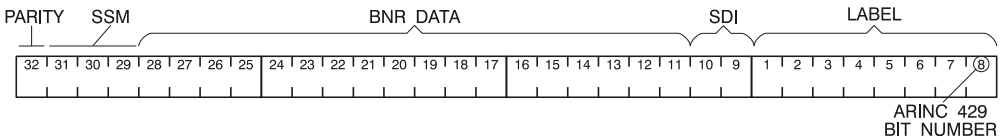
RETURNS

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

DESCRIPTION

Inserts *value* into the data field of the BNR 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 BNR and inserted into the specified bits (within bits 9 through 28). No other conversion, such as resolution or sign, is made.

The function assumes the BNR word has the following format:



The bits from the parity and SSM fields are not modified. 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). A maximum of 20 bits (when *msb*=28 and *lsb*=9) of *value* is used by this function.

SEE ALSO

BTI429_BNRGetData, BTI429_BNRPutSSM, BTI429_BNRPutSign

BNRPutMant

```
ULONG BTI429_BNRPutMant(
   ULONG msg,                  //32-bit BNR word
   ULONG value,                //New data value
   USHORT sigbit,              //Number of significant bits
   USHORT twos                 //Two's complement field
)
```

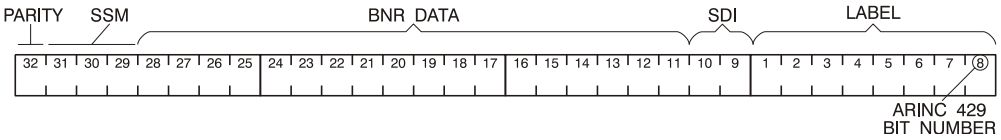
RETURNS

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

DESCRIPTION

Inserts the mantissa value into the data field of the BNR word specified in *msg*. *sigbit* specifies the number of significant bits in the BNR field. *value* is converted to BNR and inserted into the data field.

The function assumes the BNR data field is left-adjusted at bit 28 as shown below:



If *twos* is non-zero, then the two's complement of *value* is inserted into the data field, and bit 29 of the SSM field is set to one. Otherwise, bit 29 of the SSM field is set to zero.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

The *sigdig* values may range from 1 to 20 bits. No resolution scaling is done by this function. .

SEE ALSO

BTI429_BNRGetMant

BNRPutSign

```
ULONG BTI429_BNRPutSign(
   ULONG msg,                    //32-bit BNR word
   USHORT twos                   //Two's complement field
)
```

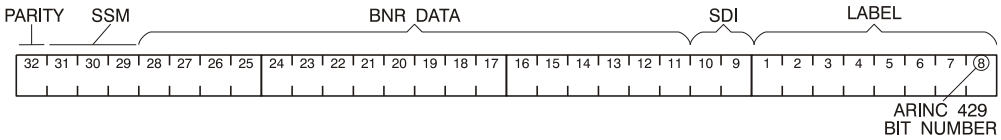
RETURNS

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

DESCRIPTION

Inserts *twos* into the SSM field of the BNR word specified in *msg*. If *twos* is non-zero, then bit 29 of the SSM field is set to one. Otherwise, bit 29 of the SSM field is set to zero.

The function assumes the SSM field is located at bits 29 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_BNRGetSign, BTI429_BNRPutSSM

BNRPutSSM

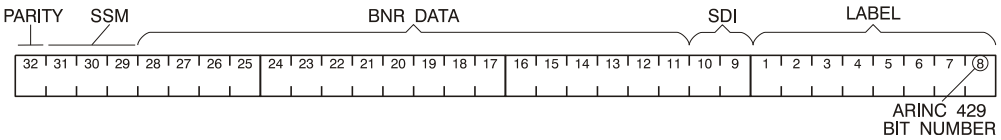
```
ULONG BTI429_BNRPutSSM(
   ULONG msg,                    //32-bit BNR word
   USHORT value                  //3-bit value of SSM field
)
```

RETURNS

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

DESCRIPTION

Inserts *value* into the 3-bit SSM field of the BNR word specified by *msg*. The function assumes the SSM field is located at bits 29 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

The sign is generally specified by bit 29. Changing this value may change the sign.

SEE ALSO

BTI429_BNRGetSSM, BTI429_BNRGetSign

BNRPutVal

```
ULONG BTI429_BNRPutVal(
   LPCSTR buf,              //Pointer to buffer containing ASCII string
   ULONG msg,               //32-bit BNR word
   USHORT sigbit,           //Number of significant bits
   LPCSTR resolstr          //Pointer to resolution string
)
```

RETURNS

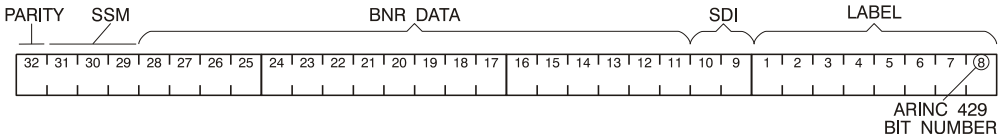
The new 32-bit BNR word with the data field inserted.

DESCRIPTION

Inserts *buf* into the data field of the BNR word specified by *msg*. *buf* holds an ASCII string containing the value to insert. This string may contain a decimal point and may be signed.

sigbit specifies the number of significant bits in the BNR data field. *resolstr* points to a null-terminated ASCII string specifying the resolution of the BNR data. This resolution string is a decimal real number that represents the value of the least significant bit (e.g., 0.25). It may contain a decimal point if needed, but should not have a sign or exponent.

The function assumes the data field is left-adjusted at bit 28 as shown below:



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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

The *sigdig* values may range from 1 to 20 bits. The sign of the result is determined by the SSM fields as described by the BTI429_BNRGetSign function.

SEE ALSO

BTI429_BNRGetVal, BTI429_BNRPutData, BTI429_BNRPutSign

CardClose

```
ERRVAL BTICard_CardClose(
   HCARD hCard                    //Card handle
)
```

RETURNS

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

DESCRIPTION

Disables access to the specified Device and releases the associated hardware resources (e.g., memory and I/O space, interrupt number, and DMA channel). BTICard_CardClose closes the Device and all of its cores opened with BTICard_CoreOpen. This function does not stop the core(s) from operating (use BTICard_CardStop to stop each core).

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

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

SEE ALSO

BTICard_CardOpen, BTICard_CoreOpen, BTICard_CardStop

CardIsRunning

```
BOOL BTICard_CardIsRunning(
   HCORE hCore                  //Core handle
)
```

RETURNS

TRUE if the core is still running, otherwise FALSE.

DESCRIPTION

Determines whether the core specified by *hCore* is running.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_CardStart, BTICard_CardStop

CardOpen

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

RETURNS

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

DESCRIPTION

Enables access to a Device. If BTICard_CardOpen finds the Device that has been assigned *cardnum*, it performs a quick hardware self-test of the Device. Since this function opens the Device and provides a card handle parameter required by BTICard_CoreOpen (which returns the core handle used by all other functions), this function is always the first BTIDriver function called by a program.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

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

SEE ALSO

BTICard_CardClose

CardProductStr

```
LPCSTR BTICard_CardProductStr(
   HCORE hCore                    //Core handle
)
```

RETURNS

A pointer to a character string describing the Device specified by *hCore*.

DESCRIPTION

Returns specific product information for the Device specified by *hCore*.

BTICard_CardTypeStr identifies the family to which a Device belongs. Inside of that family, BTICard_CardProductStr specifies product information such as model number, level or functionality, or configuration. Combine these functions to identify your Device.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_CardTypeStr

CardReset

```
VOID BTICard_CardReset(
   HCORE hCore                    //Core handle
)
```

RETURNS

None.

DESCRIPTION

Stops and performs a hardware reset on the core specified by *hCore*. If a message is being processed, the processing is allowed to finish before the core is halted.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Does not reset historic maximum and minimum Sysmon sensor values; to reset these values use BTICard_SysMonClear.

SEE ALSO

BTICard_CardStart, BTICard_CardStop, BTICard_SysMonClear

CardResume

```
ERRVAL BTICard_CardResume(
   HCORE hCore                    //Core handle
)
```

RETURNS

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

DESCRIPTION

Reactivates the specified core from the point at which it was stopped (using BTICard_CardStop). The following table compares the difference between calling BTICard_CardResume and BTICard_CardStart:

Feature	When CardStart is called….	When CardResume is called….
Transmit Schedule	Execution starts at the start of the transmit Schedule.	Execution resumes at the point the transmit Schedule was stopped.
Event Log List	Any unread entries in the Event Log List are cleared before the core is started.	Any unread records in the Event Log List are preserved as the core is resumed.
Sequential Monitor	Any unread records in the Sequential Record are cleared before the core is started.	Any unread records in the Sequential Record are preserved as the core is resumed.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTICard_CardStop must precede this function.

SEE ALSO

BTICard_CardStart, BTICard_CardStop

CardStart

```
ERRVAL BTICard_CardStart(
   HCORE hCore                    //Core handle
)
```

RETURNS

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

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

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

SEE ALSO

BTICard_CardStop, BTICard_CardIsRunning

CardStop

```
BOOL BTICard_CardStop(
   HCORE hCore                    //Core handle
)
```

RETURNS

TRUE if the core was active, otherwise FALSE.

DESCRIPTION

Stops operation of all the channel on the specified core. If a message is being processed, the processing is allowed to finish before the core is halted.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_CardStart, BTICard_CardIsRunning

CardTest

```
ERRVAL BTICard_CardTest(
   USHORT level,              //Level of tests to perform
   HCORE hCore                //Core handle
)
```

RETURNS

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

DESCRIPTION

Executes a hardware test specified by *level* on the core specified by *hCore*. When the test completes, the core is left in the same state as after a BTICard_-CardReset call.

level	
Constant	**Description**
TEST_LEVEL_0	Tests the I/O interface of the core. The test reads and writes each I/O with a walking-bit pattern.
TEST_LEVEL_1	In addition to Level 0, this level tests the memory interface of the core. The test performs a pattern test of the RAM.
TEST_LEVEL_2	In addition to previous levels, this level tests the communication process of the core. The test performs a pattern test of the RAM using the communication process.
TEST_LEVEL_3	In addition to previous levels, this level tests the encoders and decoders of the core. Where possible, the core monitors its own transmissions to validate protocol functionality.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

This function disrupts normal operation of the core. For TEST_LEVEL_3, do not use this function when the core is connected to an active databus, as the results will be unpredictable.

SEE ALSO

BTICard_CardReset

CardTrigger

```
VOID BTICard_CardTrigger (
   HCORE hCore                  //Core handle
)
```

RETURNS

None.

DESCRIPTION

Generates a software-simulated external trigger signal on all available trigger lines. For Devices with multiple trigger lines, BTICard_CardTriggerEx can be used to specify lines individually.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_CardTriggerEx, BTI429_ChTriggerDefine, BTI429_MsgTriggerDefine

CardTriggerEx

```
VOID BTICard_CardTriggerEx (
   USHORT trigmask,          //Line(s) to trigger via software
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Simulates an external trigger signal on the trigger input line(s) specified by *trigmask*. The constants in the table below may be bitwise OR-ed together to trigger multiple lines.

trigmask	
Constant	**Description**
TRIGMASK_TRIGA	Selects trigger line A
TRIGMASK_TRIGB	Selects trigger line B
TRIGMASK_TRIGC	Selects trigger line C

DEVICE DEPENDENCY

Though this function is intended for 4G and 5G Devices, which have multiple trigger lines, using a *trigmask* value of TRIGMASK_TRIGA on 3G Devices produces the same result as BTICard_CardTrigger.

WARNINGS

None.

SEE ALSO

BTICard_CardTrigger, BTI429_ChTriggerDefine, BTI429_MsgTriggerDefine

CardTypeStr

```
LPCSTR BTICard_CardTypeStr(
   HCORE hCore                    //Core handle
)
```

RETURNS

A pointer to a character string describing the Device specified by *hCore*.

DESCRIPTION

Returns the type of Device specified by *hCore*.

BTICard_CardTypeStr identifies the family to which a Device belongs. Inside of that family, BTICard_CardProductStr specifies product information such as model number, level of functionality, or configuration. Combine these functions to identify your Device.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_CardProductStr

ChClear

```
ERRVAL BTI429_ChClear(
   INT channel,              //Channel number
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Clears the contents of the specified channel. If the channel is a transmitter, all Command Blocks in the transmit Schedule are deleted. If the channel is a receiver, all filters are deleted. The contents of the Message Records are unaffected. The configuration options previously set by BTI429_ChConfig are unchanged.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_CardReset, BTI429_ChConfig

ChConfig

```
ERRVAL BTI429_ChConfig(
   ULONG ctrlflags,          //Selects channel options
   INT channel,              //Number of channel
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

1. Stops the channel.
2. Clears transmit Schedule for the specified channel (Filter Tables are not affected).
3. Writes Device options defined by *ctrlflags*.
4. Restarts the channel if previously started and not disabled by CHCFG429_INACTIVE.

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
CHCFG429_DEFAULT	Select all default settings (**bold** below)	✓	✓
CHCFG429_LOWSPEED	Select low speed messages (12.5 Kbps)	✓	✓
CHCFG429_HIGHSPEED	Select high speed messages (100 Kbps)	✓	✓
CHCFG429_AUTOSPEED	Select auto speed detection of messages	✓	
CHCFG429_PARODD	Select odd parity	✓	✓
CHCFG429_PAREVEN	Select even parity	✓	✓
CHCFG429_PARDATA	Select parity bit as data (ignores parity)	✓	✓
CHCFG429_ACTIVE	Enable channel activity	✓	✓
CHCFG429_INACTIVE	Disable channel activity	✓	✓
CHCFG429_SEQSEL	Sequential monitoring selected at message level	✓	✓
CHCFG429_SEQALL	Every message will be recorded in the Sequential Record	✓	✓
CHCFG429_NOLOGHALT	No entry will be made in the Event Log List on a HALT command		✓
CHCFG429_LOGHALT	An entry will be made in the Event Log List on a HALT command		✓
CHCFG429_NOLOGPAUSE	No entry will be made in the Event Log List on a PAUSE command		✓
CHCFG429_LOGPAUSE	An entry will be made in the Event Log List on a PAUSE command		✓
CHCFG429_NOLOGERR	No entry will be made in the Event Log List when a de-coder detects an error	✓	
CHCFG429_LOGERR	An entry will be made in the Event Log List when a de-coder detects an error	✓	
CHCFG429_TIMETAGOFF	The time-tag option is selected at the message level	✓	✓
CHCFG429_TIMETAG	All messages will record a time-tag.	✓	✓
CHCFG429_ELAPSEOFF	The elapse timing option is selected at the message level	✓	✓
CHCFG429_ELAPSE	All messages will record an elapsed time.	✓	✓
CHCFG429_MAXMINOFF	Max and min repetition rates are selected at the message level	✓	✓
CHCFG429_MAX	All messages will record a max time	✓	✓
CHCFG429_MIN	All messages will record a min time	✓	✓
CHCFG429_MAXMIN	All messages will record a max and a min time	✓	✓

(Continued on next page)

(Continued from previous page)

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
CHCFG429_NOHIT	The Hit Counter is selected at the message level	✓	✓
CHCFG429_HIT	The Hit Counter is selected for all messages (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
CHCFG429_SELFTESTOFF	This channel will transmit/receive on the operational bus and not the self-test bus.	✓	✓
CHCFG429_SELFTEST	This channel will transmit/receive on the internal self-test bus and not the operational bus. Only one transmitter can be on the self-test bus at a time; therefore, only the last transmit channel configured to use the self-test bus will use this option.	✓	✓
CHCFG429_SYNCOFF	The SYNCOUT signal option will be selected at the message level	✓	✓
CHCFG429_SYNC	The SYNCOUT signal will be active for all messages of this channel	✓	✓
CHCFG429_EXTOFF	Messages are selected for an external trigger at the message level		✓
CHCFG429_EXTTRIG	All messages for this channel will be externally triggered		✓
CHCFG429_NOERR	Error injection is selected at the message level		✓
CHCFG429_PARERR	All messages will have a parity error (inverts the parity bit)		✓
CHCFG429_UNPAUSE	The channel will initially be unpaused	✓	✓
CHCFG429_PAUSE	The channel will initially be paused	✓	✓
CHCFG429_NOLOOPMAX	Disable Schedule maximum loop count		✓
CHCFG429_LOOPMAX	Enable Schedule maximum loop count		✓

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

5G Devices do not support the CHCFG429_LOOPMAX flag.

WARNINGS

The function clears any previous configuration of the channel.

SEE ALSO

BTICard_CardStart, BTICard_CardStop

ChConfigEx

```
ERRVAL BTI429_ChConfigEx(
   ULONG ctrlflags,          //Selects channel options
   USHORT count,             //Number of entries in the schedule
   INT channel,              //Number of channel
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Configures the channel similar to BTI429_ChConfig, but with the addition of the parameter *count* which is used to specify the number of schedule entries to allocate for the schedule. BTI429_ChConfig defaults to 512 schedule entries, and *count* can be used to allocate a different number.

DEVICE DEPENDENCY

3G and 4G Devices support up to 4089 schedule entries while 5G Devices support up to 8187 schedule entries.

WARNINGS

The function clears any previous configuration of the channel.

SEE ALSO

BTICard_CardStart, BTICard_CardStop

ChGetCount

```
VOID BTI429_ChGetCount(
   LPINT rcvcount,          //Pointer to variable to hold receiver count
   LPINT xmtcount,          //Pointer to variable to hold transmitter count
   HCORE hCore              //Core handle
)
```

RETURNS

None.

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_ChIsRcv, BTI429_ChIsXmt. BTI429_ChIs429, BTI429_ChGetInfo

ChGetInfo

```
ULONG BTI429_ChGetInfo(
   USHORT infotype,          //Type of information to be returned
   INT channel,              //Channel number
   HCORE hCore               //Core handle
)
```

RETURNS

The requested information about the specified channel.

DESCRIPTION

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

infotype		
Constant	**Returns**	**Description**
INFO429_PARAM	1=TRUE 0=FALSE	Channel is parametric Channel is not parametric

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_ParamAmplitudeConfig, BTI429_ParamBitRateConfig

ChIs429

```
BOOL BTI429_ChIs429(
   INT channel,                  //Channel number to test
   HCORE hCore                   //Core handle
)
```

RETURNS

TRUE if the channel is ARINC 429, otherwise FALSE.

DESCRIPTION

Checks to see if the channel number specified by *channel* is a 429 channel.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_ChGetInfo, BTI429_ChIsRcv, BTI429_ChIsXmt, BTI429_ChGet-Count

ChIsRcv

```
BOOL BTI429_ChIsRcv (
   INT channel,              //Channel number to test
   HCORE hCore               //Core handle
)
```

RETURNS

TRUE if the channel is an ARINC 429 receiver, or FALSE if it is not a receiver.

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

If this function returns a value of zero, do not assume that the channel must then be a transmitter, because the channel may not exist at all. A call to BTI-429_ChIsXmt must be made to be sure that the channel is a transmitter.

SEE ALSO

BTI429_ChIsXmt, BTI429_ChGetCount, BTI429_ChIs429

ChIsXmt

```
BOOL BTI429_ChIsXmt (
   INT channel,                  //Channel number to test
   HCORE hCore                   //Core handle
)
```

RETURNS

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

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

If this function returns a value of zero, do not assume that the channel must then be a receiver, because the channel may not exist at all. A call to BTI429_-ChIsRcv must be made to be sure that the channel is a receiver.

SEE ALSO

BTI429_ChIsRcv, BTI429_ChGetCount, BTI429_ChIs429

ChPause

```
VOID BTI429_ChPause(
   INT channel,              //Channel number
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Pauses the operation of the channel specified by *channel*. All activity on the channel ceases. The channel remains paused until the channel is resumed by BTI429_ChResume. BTI429_ChConfig initializes the channel as either un-paused (default) or paused.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Do not confuse this channel pause with either channel enable or the card-level controls. Channel enable is controlled by BTI429_ChConfig, BTI429_ChStart, and BTI429_ChStop. Card-level controls are activated through BTICard_-CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTI429_ChResume, BTI429_SchedPause

ChPauseCheck

```
INT BTI429_ChPauseCheck(
   INT channel,                  //Channel number
   HCORE hCore                   //Core handle
)
```

RETURNS

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

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_ChPause, BTI429_ChResume

ChResume

```
VOID BTI429_ChResume(
   INT channel,                  //Channel number
   HCORE hCore                   //Core handle
)
```

RETURNS

None.

DESCRIPTION

Resumes the operation of the channel specified by *channel* after it has been paused by BTI429_ChPause or the Device has encountered a PAUSE Command Block in the transmit Schedule. BTI429_ChConfig initializes the channel as either unpaused (default) or paused. If the Device is running, all activity on the channel will begin. If the Device is stopped, channel activity will begin when the Device is started.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Do not confuse this channel pause with either channel enable or the card-level controls. Channel enable is controlled by BTI429_ChConfig, BTI429_ChStart, and BTI429_ChStop. Card-level controls are activated through BTICard_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTI429_ChPause, BTI429_SchedPause

ChStart

```
BOOL BTI429_ChStart(
   INT channel,                  //Channel number
   HCORE hCore                   //Core handle
)
```

RETURNS

TRUE if the channel was previously enabled, otherwise FALSE.

DESCRIPTION

Enables the operation of the channel specified by *channel*. If it is a transmit channel, the Schedule restarts at the beginning. The channel remains enabled until BTI429_ChStop is called or a HALT Command Block is encountered in the transmit Schedule. If the Device is stopped, then channel activity begins when the Device is started with BTICard_CardStart.

BTI429_ChStart and BTI429_ChStop enable and disable a channel. BTI-429_ChConfig initializes the channel as either enabled (default) or disabled. These functions allow the channel to be stopped and reconfigured with different settings while other channels are running.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Do not confuse this channel enable with either channel pause or the card-level controls. Channel pause is controlled by BTI429_ChConfig, BTI429_ChPause, and BTI429_ChResume. Card-level controls are activated through BTI-Card_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTI429_ChStop, BTICard_CardStart, BTICard_CardStop, BTI429_SchedHalt

ChStop

```
BOOL BTI429_ChStop(
   INT channel,                  //Channel number
   HCORE hCore                   //Core handle
)
```

RETURNS

TRUE if the channel was previously enabled, otherwise FALSE.

DESCRIPTION

Disables operation of the channel specified by *channel*. If a message is being sent or received, the processing is allowed to finish before the channel is halted. Use BTI429_ChStart to reenable the channel.

BTI429_ChStart and BTI429_ChStop enable and disable a channel. BTI-429_ChConfig initializes the channel as either enabled (default) or disabled. These functions allow the channel to be stopped and reconfigured with different settings while other channels are running.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Do not confuse this channel enable with either channel pause or the card-level controls. Channel pause is controlled by BTI429_ChConfig, BTI429_ChPause, and BTI429_ChResume. Card-level controls are activated through BTI-Card_CardStart, BTICard_CardStop, and BTICard_CardResume.

SEE ALSO

BTI429_ChStart, BTI429_SchedHalt, BTICard_CardStart, BTICard_CardStop

ChSyncDefine

```
ERRVAL BTI429_ChSyncDefine(
   BOOL enable,              //Enable/disable external sync output
   USHORT syncmask,          //Line(s) used for sync output
   USHORT pinpolarity,       //Active pin polarity (high/low)
   INT channel,              //Channel number (transmit or receive)
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Defines the sync output settings for *channel* on *hCore* and configures all messages to output a sync signal. This sync signal appears on the line(s) specified by *syncmask* with the polarity specified by *pinpolarity* (see tables below). When enabled, the sync signal is active during the transmission or reception of the message. BTI429_ChSyncDefine may be called during run time to redefine the sync output settings.

The constants in the tables below may be bitwise OR-ed together to configure multiple lines.

syncmask	
Constant	**Description**
SYNCMASK_SYNCA	Selects sync line A
SYNCMASK_SYNCB	Selects sync line B
SYNCMASK_SYNCC	Selects sync line C

pinpolarity	
Constant	**Description**
SYNCPOL_SYNCAL	Sets active low polarity for sync line A
SYNCPOL_SYNCAH	Sets active high polarity for sync line A
SYNCPOL_SYNCBL	Sets active low polarity for sync line B
SYNCPOL_SYNCBH	Sets active high polarity for sync line B
SYNCPOL_SYNCCL	Sets active low polarity for sync line C
SYNCPOL_SYNCCH	Sets active high polarity for sync line C

Alternatively, to configure selected transmit message(s) to output a sync pulse, use BTI429_MsgSyncDefine.

DEVICE DEPENDENCY

Applies to 4G and 5G Devices. 3G Devices, which have a single sync line, can call BTI429_ChConfig with the CHCFG429_SYNC flag. The mapping of sync lines to discretes is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTI429_MsgSyncDefine

ChTriggerDefine

```
ERRVAL BTI429_ChTriggerDefine(
   BOOL enable,              //Enable/disable external trigger
   USHORT trigmask,          //Line(s) used for trigger signal
   USHORT trigval,           //Active/inactive condition for trigger line(s)
   USHORT pinpolarity,       //Active pin polarity (high/low)
   INT xmtchan,              //Transmit channel number
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Defines the transmit trigger settings for *xmtchan* on *hCore* and configures every message in the transmit schedule to wait for the defined trigger signal before transmitting. The input line(s) are specified by *trigmask* with an active trigger state being the combination of *trigval* and *pinpolarity* (see tables below). BTI-429_ChTriggerDefine may be called during run time to redefine the trigger input settings.

The constants in the tables below may be bitwise OR-ed together to configure multiple lines. All combined states must be true for the trigger to occur.

trigmask	
Constant	**Description**
TRIGMASK_TRIGA	Selects trigger line A
TRIGMASK_TRIGB	Selects trigger line B
TRIGMASK_TRIGC	Selects trigger line C

trigval	
Constant	**Description**
TRIGVAL_TRIGAOFF	Trigger on line A inactive
TRIGVAL_TRIGAON	Trigger on line A active
TRIGVAL_TRIGBOFF	Trigger on line B inactive
TRIGVAL_TRIGBON	Trigger on line B active
TRIGVAL_TRIGCOFF	Trigger on line C inactive
TRIGVAL_TRIGCON	Trigger on line C active

pinpolarity	
Constant	**Description**
TRIGPOL_TRIGAL	Sets active low polarity for trigger line A
TRIGPOL_TRIGAH	Sets active high polarity for trigger line A
TRIGPOL_TRIGBL	Sets active low polarity for trigger line B
TRIGPOL_TRIGBH	Sets active high polarity for trigger line B
TRIGPOL_TRIGCL	Sets active low polarity for trigger line C
TRIGPOL_TRIGCH	Sets active high polarity for trigger line C

Alternatively, to associate selected message(s) to a trigger signal, use BTI429_-MsgTriggerDefine.

DEVICE DEPENDENCY

Applies to 4G and 5G Devices. 3G Devices, which have a single external trigger, can call BTI429_ChConfig with the CHCFG429_EXTTRIG flag. The mapping of trigger lines to discretes is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTI429_MsgTriggerDefine, BTICard_CardTrigger, BTICard_CardTriggerEx

CmdShotRd

```
BOOL BTI429_CmdShotRd(
   SCHNDX index,              //Schedule index of item to read
   INT channel,               //Channel number
   HCORE hCore                //Core handle
)
```

RETURNS

TRUE if the single-shot bit is set, otherwise FALSE if not set.

DESCRIPTION

Reads the value of the single-shot bit for the schedule opcode specified by *index*.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Some scheduling functions insert more than one opcode into the schedule. Therefore, it may be necessary to call this function on multiple schedule index values to get the desired effect.

SEE ALSO

BTI429_CmdShotWr

CmdShotWr

```
ERRVAL BTI429_CmdShotWr(
   BOOL value,                  //Value of single-shot bit
   SCHNDX index,                //Schedule index of item to single-shot
   INT channel,                 //Channel number
   HCORE hCore                  //Core handle
)
```

RETURNS

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

DESCRIPTION

Sets the single-shot bit to *value* for the schedule entry specified by *index*. When set to TRUE, the single-shot bit instructs the schedule to process the specified opcode one time, and then to set the skip bit after processing is complete. The single-shot bit is FALSE (disabled) by default.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Some scheduling functions insert more than one opcode into the schedule. Therefore, it may be necessary to call this function on multiple schedule index values to get the desired effect.

SEE ALSO

BTI429_CmdShotRd

CmdSkipRd

```
BOOL BTI429_CmdSkipRd(
   SCHNDX index,              //Schedule index of item to read
   INT channel,               //Channel number
   HCORE hCore                //Core handle
)
```

RETURNS

TRUE if the skip bit is set, otherwise FALSE if not set.

DESCRIPTION

Reads the value of the skip bit for the schedule opcode specified by *index*.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Some scheduling functions insert more than one opcode into the schedule. Therefore, it may be necessary to call this function on multiple schedule index values to get the desired effect.

SEE ALSO

BTI429_CmdSkipWr

CmdSkipWr

```
ERRVAL BTI429_CmdSkipWr(
   BOOL value,                  //Value of skip bit
   SCHNDX index,                //Schedule index of item to skip
   INT channel,                 //Channel number
   HCORE hCore                  //Core handle
)
```

RETURNS

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

DESCRIPTION

Sets the skip bit to *value* for the schedule entry specified by *index*. When set to TRUE, the skip bit instructs the schedule to skip over processing the specified opcode. The skip bit is FALSE (disabled) by default.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Some scheduling functions insert more than one opcode into the schedule. Therefore, it may be necessary to call this function on multiple schedule index values to get the desired effect.

SEE ALSO

BTI429_CmdSkipRd

CmdStepRd

```
BOOL BTI429_CmdStepRd(
   SCHNDX index,              //Schedule index of item to read
   INT channel,               //Channel number
   HCORE hCore                //Core handle
)
```

RETURNS

TRUE if the step bit is set, otherwise FALSE if not set.

DESCRIPTION

Reads the value of the step bit for the schedule opcode specified by *index*.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Some scheduling functions insert more than one opcode into the schedule. Therefore, it may be necessary to call this function on multiple schedule index values to get the desired effect.

The user is especially cautioned to consider the effects of stepping through op-codes that depend upon or generate hardware timing.

SEE ALSO

BTI429_CmdStepWr

CmdStepWr

```
ERRVAL BTI429_CmdStepWr(
   BOOL value,                //Value of step bit
   SCHNDX index,              //Schedule index of item to step
   INT channel,               //Channel number
   HCORE hCore                //Core handle
)
```

RETURNS

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

DESCRIPTION

Sets the step bit to *value* for the schedule entry specified by *index*. When set to TRUE, the step bit instructs the schedule to pause after processing the specified opcode. The step bit is FALSE (disabled) by default.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Some scheduling functions insert more than one opcode into the schedule. Therefore, it may be necessary to call this function on multiple schedule index values to get the desired effect.

The user is especially cautioned to consider the effects of stepping through op-codes that depend upon or generate hardware timing.

SEE ALSO

BTI429_CmdStepRd

CoreOpen

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

RETURNS

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

DESCRIPTION

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

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

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

SEE ALSO

BTICard_CardOpen, BTICard_CardClose

ErrDescStr

```
LPCSTR BTICard_ErrDescStr(
   ERRVAL errval,            //An error value
   HCORE hCore               //Core handle
)
```

RETURNS

A pointer to a character string describing the error specified by *errval*.

DESCRIPTION

Describes the error value specified by *errval*.

All functions of type ERRVAL return a negative value if an error occurs, or zero if successful. BTICard_ErrDescStr returns a description of the specified error value.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

ERRVAL type functions

EventLogConfig

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

RETURNS

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

DESCRIPTION

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

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

DEVICE DEPENDENCY

The size of the Event Log for 5G Devices is always 256 entries regardless of *count*.

WARNINGS

None.

SEE ALSO

BTICard_EventLogRd, BTICard_EventLogStatus

EventLogRd

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

RETURNS

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

DESCRIPTION

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

The value of *typeval* determines the meaning of the *infoval* value (see table below). Note that the Event Log List records events from all protocols implemented on the specified core. The first two event types, regarding the Sequential Record, are protocol-independent. The rest of the table is subdivided by protocol.

	typeval	**Description**	***infoval***	**Refer to…**
General	EVENTTYPE_SEQFULL	Sequential Record full (halted) or overwritten	Address of last entry	BTICard_SeqConfig
	EVENTTYPE_SEQFREQ	n^{th} entry (user-specified)	Address of last entry	BTICard_SeqConfig
	EVENTTYPE_BITERROR	BIT system detected an error	BIT Status at the time of the Error	BTICard_BITConfig
MIL-STD-1553	EVENTTYPE_1553MSG	Message processed	Address of the Message structure	BTI1553_BCCreateMsg BTI1553_RTCreateMsg BTI1553_BCCreateList BTI1553_RTCreateList
	EVENTTYPE_1553OPCODE	BC Schedule encountered a LOG command	User-assigned tag value	BTI1553_BCSchedLog
	EVENTTYPE_1553HALT	BC Schedule encountered a HALT command	Address of the Schedule entry	BTI1553_BCSchedHalt
	EVENTTYPE_1553PAUSE	BC Schedule encountered a PAUSE command	Address of the Schedule entry	BTI1553_BCSchedPause
	EVENTTYPE_1553LIST	List Buffer empty/ full (underflow or overflow)	List Buffer address	BTI1553_BCCreateList BTI1553_RTCreateList

(Continued on next page)

(Continued from previous page)

	typeval	Description	*infoval*	Refer to…
ARINC 429	EVENTTYPE_429MSG	Message received or transmitted	Address of the Message Record	BTI429_MsgCreate BTI429_FilterDefault BTI429_FilterSet
	EVENTTYPE_429OPCODE	A transmit Schedule encountered a LOG command	User-assigned tag value	BTI429_SchedLog
	EVENTTYPE_429HALT	A transmit Schedule encountered a HALT command	Address of the Schedule entry	BTI429_ChConfig BTI429_SchedHalt
	EVENTTYPE_429PAUSE	A transmit Schedule encountered a PAUSE command	Address of the Schedule entry	BTI429_ChConfig BTI429_SchedPause
	EVENTTYPE_429LIST	List Buffer empty or full (underflow or overflow)	List Buffer address	BTI429_ListAsyncCreate BTI429_ListRcvCreate BTI429_ListXmtCreate
	EVENTTYPE_429ERR	A decoder error was detected	Address of the message that contained the error	BTI429_ChConfig
ARINC 717	EVENTTYPE_717WORD	Processed 717 word	Word address	BTI717_SubFrmWordConfig BTI717_SuperFrmWordConfig
	EVENTTYPE_717SUBFRM	Processed 717 subframe	Subframe number	BTI717_SubFrmWordConfig BTI717_SuperFrmWordConfig
	EVENTTYPE_717SYNCERR	717 receive channel lost sync	Channel number	BTI717_ChConfig
ARINC 708	EVENTTYPE_708MSG	Message received/ transmitted	Message index	BTI708_RcvBuild BTI708_XmtBuild

DEVICE DEPENDENCY

The value passed back in *channel* may not be valid for all types of events. If an event does not have an associated channel value, it is filled in with the value FFh.

WARNINGS

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

SEE ALSO

BTICard_EventLogConfig

EventLogStatus

```
INT BTICard_EventLogStatus(
   HCORE hCore                    //Core handle
)
```

RETURNS

The status value of the Event Log List.

DESCRIPTION

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

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

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

SEE ALSO

BTICard_EventLogConfig, BTICard_EventLogRd

ExtDIOMonConfig

```
ERRVAL BTICard_ExtDIOMonConfig(
   USHORT risemask,          //Discrete bitmask to monitor rising edges
   USHORT fallmask,          //Discrete bitmask to monitor falling edges
   INT banknum,              //Bank number of DIO (should be set to 0)
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Enables Sequential Monitoring on specific transitions of discrete inputs. Discrete inputs are sampled at a minimum rate of 100 KHz and compared to the previously sampled values. If the digital value of any discretes specified in the *risemask* have transitioned from a zero to a one, or the digital value of any discretes specified in the *fallmask* have transitioned from a one to a zero, then a Sequential Record will be created. To disable previously-enabled monitor settings, call BTICard_ExtDIOMonConfig again with *risemask* and *fallmask* both set to zero.

The *risemask* and *fallmask* are a bit mask specifying up to 16 discrete input signals. The LSB of each value corresponds to *dionum* 1 and the MSB corresponds to *dionum* 16.

DEVICE DEPENDENCY

Applies to 5G Devices. The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

By default, avionics discretes are active low. A grounded avionics discrete will have a digital value of one. If the polarity for that discrete input has been changed, the digital value compared for rising and falling edges will be reversed.

SEE ALSO

BTICard_SeqFindNextDIO

ExtDIORd

```
BOOL BTICard_ExtDIORd(
   INT dionum,                  //Specifies the DIO number
   HCORE hCore                  //Core handle
)
```

RETURNS

Status of the digital I/O pin. Returns a zero if the pin is inactive or a one if the pin is active.

DESCRIPTION

Reads the status of the digital I/O pin specified by *dionum*.

DEVICE DEPENDENCY

The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_ExtDIOWr, BTI429_SchedPulse

ExtDIOWr

```
VOID BTICard_ExtDIOWr(
   INT dionum,              //Specifies the DIO number
   BOOL dioval,             //The value to set
   HCORE hCore              //Core handle
)
```

RETURNS

None.

DESCRIPTION

Sets the digital I/O pin specified by *dionum* to the value specified by *dioval*. A *dioval* of zero sets the pin to inactive, and a *dioval* of one sets the pin to active.

DEVICE DEPENDENCY

The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

Some products use avionics discretes while others use digital I/O discretes. When using the digital I/O as an output (as this function does), do not drive the digital I/O pin from an external source as this may damage the Device. Please consult the hardware manual for the Device.

SEE ALSO

BTICard_ExtDIORd, BTI429_SchedPulse

ExtLEDRd

```
BOOL BTICard_ExtLEDRd(
   HCORE hCore                    //Core handle
)
```

RETURNS

Returns a zero if the LED is off, or a one if the LED is on.

DESCRIPTION

Reads the state of the on-board LED.

DEVICE DEPENDENCY

4G Devices have a user-controlled LED for each core. For all other Devices, please refer to the hardware manual.

WARNINGS

None.

SEE ALSO

BTICard_ExtLEDWr

ExtLEDWr

```
VOID BTICard_ExtLEDWr(
   BOOL ledval,              //New state of the LED
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Sets the state of the on-board LED. An *ledval* of zero turns the LED off, and an *ledval* of one turns the LED on.

DEVICE DEPENDENCY

4G Devices have a user-controlled LED for each core. For all other Devices, please refer to the hardware manual.

WARNINGS

None.

SEE ALSO

BTICard_ExtLEDRd

ExtStatusLEDRd

```
VOID BTICard_ExtStatusLEDRd(
   LPINT ledval,              //Pointer to variable to receive LED state
   LPINT ledcolor,            //Pointer to variable to receive LED color
   HCORE hCore                //Core handle
)
```

RETURNS

None.

DESCRIPTION

Reads the state of the on-board Status LED. A zero value will be passed to *ledval* if the LED is off, and a one value if the LED is on.

DEVICE DEPENDENCY

4G Devices have a red status LED.

BUSBox BB1xxx Devices have a multi-color Status LED. The color state of that LED can be read through *ledcolor*. A zero value indicates a red color, and a one value indicates a green color.

5G USB Adapter Devices have a red Status LED.

For all other Devices, please refer to the Device specific hardware manual.

WARNINGS

None.

SEE ALSO

BTICard_ExtStatusLEDWr, BTICard_ExtLEDRd

ExtStatusLEDWr

```
VOID BTICard_ExtStatusLEDWr(
   BOOL ledval,              //New state of the LED
   BOOL ledcolor,            //New color of the LED
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Sets the state of the on-board Status LED. An *ledval* of zero turns the LED off, and an *ledval* of one turns the LED on.

DEVICE DEPENDENCY

4G Devices have a red status LED.

BUSBox BB1xxx Devices have a multi-color Status LED. The color state of that LED can be controlled through *ledcolor*. A zero value indicates a red color, and a one value indicates a green color.

RPC Devices have a red Status LED that indicates a succesful booting of the Device. Afterwards, the Status LED can be controlled with this function.

5G USB Adapter Devices have a red Status LED.

For all other Devices, please refer to the Device specific hardware manual.

WARNINGS

None.

SEE ALSO

BTICard_ExtStatusLEDRd, BTICard_ExtLEDWr

FilterDefault

```
MSGADDR BTI429_FilterDefault(
   ULONG ctrlflags,          //Selects message options
   INT channel,              //Number of receive channel
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

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

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRT429_DEFAULT	Select all default settings (**bold** below)	✓	✓
MSGCRT429_NOSEQ	This message will not get recorded in the Sequential Record	✓	✓
MSGCRT429_SEQ	This message will get recorded in the Sequential Record	✓	✓
MSGCRT429_NOLOG	This message will not create an entry in the Event Log List	✓	✓
MSGCRT429_LOG	This message will create an entry in the Event Log List	✓	✓
MSGCRT429_NOTIMETAG	This message will not record a time-tag	✓	✓
MSGCRT429_TIMETAG	This message will record a time-tag	✓	✓
MSGCRT429_NOELAPSE	This message will not record an Elapsed Time	✓	✓
MSGCRT429_ELAPSE	This message will record an Elapsed Time	✓	✓
MSGCRT429_NOMAXMIN	This message will not record maximum and minimum repetition rates	✓	✓
MSGCRT429_MAX	This message will record maximum repetition rates	✓	✓
MSGCRT429_MIN	This message will record minimum repetition rates	✓	✓
MSGCRT429_MAXMIN	This message will record maximum and minimum repetition rates	✓	✓
MSGCRT429_NOHIT	This message will not record a Hit Counter	✓	✓
MSGCRT429_HIT	This message will record a Hit Counter (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
MSGCRT429_NOSKIP	This message will not be skipped	✓	✓
MSGCRT429_SKIP	This message will be skipped, and none of the options will be processed	✓	✓
MSGCRT429_NOSYNC	This message will not generate a SYNCOUT signal		✓
MSGCRT429_SYNC	This message will generate a SYNCOUT signal		✓
MSGCRT429_NOEXTRIG	This message will be triggered immediately		✓
MSGCRT429_EXTRIG	This message will wait for an EXTRIG* pulse to be triggered		✓
MSGCRT429_NOERR	This message will not have a parity error		✓
MSGCRT429_PARERR	This message will have a parity error		✓

(Continued on next page)

(Continued from previous page)

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRT429_WIPE	The data fields of this message will initially be wiped to a value	✓	✓
MSGCRT429_NOWIPE	The data fields of this message will initially be left unchanged	✓	✓
MSGCRT429_WIPE0	The data fields of this message will be wiped with a value of zeros (this option does not get used if MSGCRT429_NOWIPE is used)	✓	✓
MSGCRT429_WIPE1	The data fields of this message will be wiped with a value of ones (this option does not get used if MSGCRT429_NOWIPE is used)	✓	✓

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

This function initializes all filters (all label/SDI combinations) for the specified channel. Any filters previously created for the channel are overwritten. Therefore, BTI429_FilterDefault MUST precede any calls to BTI429_FilterSet.

SEE ALSO

BTI429_FilterSet, BTI429_FilterRd, BTI429_FilterWr

FilterRd

```
MSGADDR BTI429_FilterRd(
   INT label,                  //Label value
   INT sdi,                    //SDI pattern
   INT channel,                //Number of receive channel
   HCORE hCore                 //Core handle
)
```

RETURNS

Address of the Message Record pointed to by the Filter for the given parameters.

DESCRIPTION

Reads the address of the Message Record pointed to by the Filter Table for the specified *channel*, *label*, and *sdi* values.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

This value reads the address of the Message Record that a filter is pointing to, not the ARINC 429 data word. Use BTI429_MsgDataRd to read the ARINC 429 data from a Message Record.

SEE ALSO

BTI429_FilterWr, BTI429_FilterSet, BTI429_FilterDefault, BTI429_MsgDataRd

FilterSet

```
MSGADDR BTI429_FilterSet(
   ULONG ctrlflags,          //Selects message options
   INT label,                //Label value to receive
   INT sdi,                  //SDI patterns to receive
   INT channel,              //Number of receive channel
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

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

sdi allows one or more SDI combinations to be specified. A filter is created for each specified SDI. The predefined constants listed below can be used to specify the SDI. When a combination of SDIs are selected, the constants should be OR-ed together.

sdi	
Constants	**Description**
SDI00	Selects only the SDI value of 00.
SDI01	Selects only the SDI value of 01.
SDI10	Selects only the SDI value of 10.
SDI11	Selects only the SDI value of 11.
SDIALL	Selects all the SDI values.

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

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRT429_DEFAULT	Select all default settings (**bold** below)	✓	✓
MSGCRT429_NOSEQ	This message will not get recorded in the Sequential Record	✓	✓
MSGCRT429_SEQ	This message will get recorded in the Sequential Record	✓	✓
MSGCRT429_NOLOG	This message will not create an entry in the Event Log List	✓	✓
MSGCRT429_LOG	This message will create an entry in the Event Log List	✓	✓
MSGCRT429_NOTIMETAG	This message will not record a time-tag	✓	✓
MSGCRT429_TIMETAG	This message will record a time-tag	✓	✓
MSGCRT429_NOELAPSE	This message will not record an Elapsed Time	✓	✓
MSGCRT429_ELAPSE	This message will record an Elapsed Time	✓	✓

(Continued on next page)

(Continued from previous page)

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRT429_NOMAXMIN	This message will not record maximum and minimum repetition rates	✓	✓
MSGCRT429_MAX	This message will record maximum repetition rates	✓	✓
MSGCRT429_MIN	This message will record minimum repetition rates	✓	✓
MSGCRT429_MAXMIN	This message will record maximum and minimum repetition rates	✓	✓
MSGCRT429_NOHIT	This message will not record a Hit Counter	✓	✓
MSGCRT429_HIT	This message will record a Hit Counter (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
MSGCRT429_NOSKIP	This message will not be skipped	✓	✓
MSGCRT429_SKIP	This message will be skipped, and none of the options will be processed	✓	✓
MSGCRT429_NOSYNC	This message will not generate a SYNCOUT signal		✓
MSGCRT429_SYNC	This message will generate a SYNCOUT signal		✓
MSGCRT429_NOEXTRIG	This message will be triggered immediately		✓
MSGCRT429_EXTRIG	This message will wait for an EXTRIG* pulse to be triggered		✓
MSGCRT429_NOERR	This message will not have a parity error		✓
MSGCRT429_PARERR	This message will have a parity error		✓
MSGCRT429_WIPE	The data fields of this message will initially be wiped to a value	✓	✓
MSGCRT429_NOWIPE	The data fields of this message will initially be left unchanged	✓	✓
MSGCRT429_WIPE0	The data fields of this message will be wiped with a value of zeros (this option does not get used if MSGCRT429_NOWIPE is used)	✓	✓
MSGCRT429_WIPE1	The data fields of this message will be wiped with a value of ones (this option does not get used if MSGCRT429_NOWIPE is used)	✓	✓

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

None.

SEE ALSO

BTI429_FilterDefault, BTI429_FilterRd, BTI429_FilterWr

FilterWr

```
ERRVAL BTI429_FilterWr(
   MSGADDR message,          //Message address to write to filter
   INT label,                //Label value
   INT sdi,                  //SDI value
   INT channel,              //Number of receive channel
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Writes the message address (*message*) of the Message Record into the Filter Table position specified by *channel*, *label*, and *sdi*. The valid range of *sdi* is 0-3.

This function is most useful to assign multiple labels to point to one Message Record. After calling BTI429_FilterSet for the first label of interest, this function could be called with the message address that was returned by BTI429_FilterSet for each of the remaining labels. This would point all the labels of interest to one Message Record.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_FilterRd, BTI429_FilterSet, BTI429_FilterDefault

FldGetData

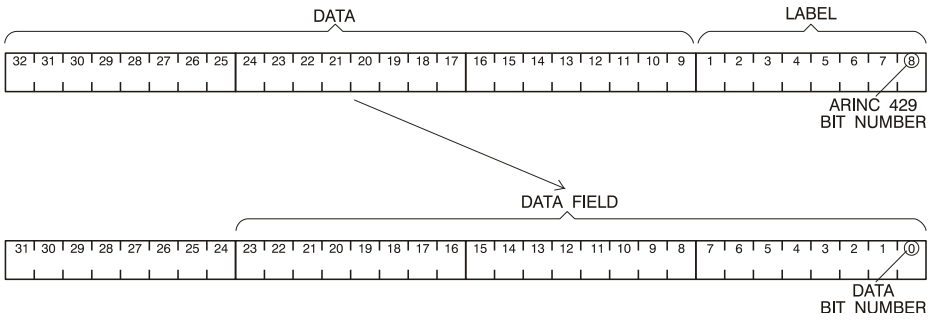
```
ULONG BTI429_FldGetData(
   ULONG msg                    //32-bit ARINC 429 word
)
```

RETURNS

The 24-bit field including both the data and parity of an ARINC 429 word.

DESCRIPTION

Extracts the 23-bit data and 1-bit parity fields of the ARINC 429 word in *msg*. The extracted 24-bit field is right-shifted and zero-filled 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_FldPutData

FldGetLabel

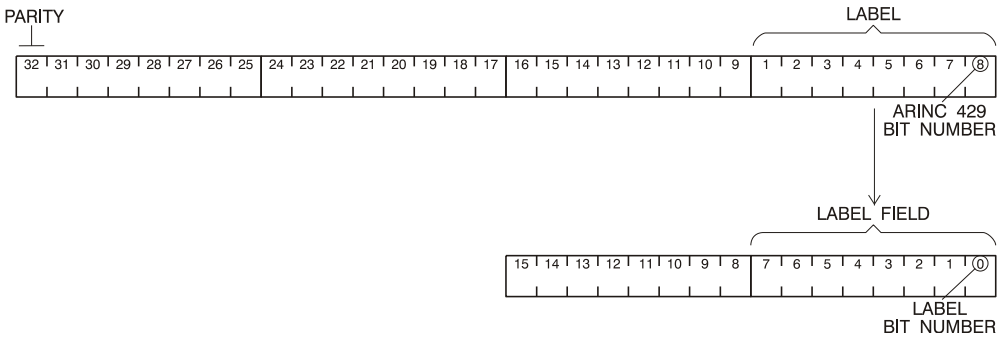
```
USHORT BTI429_FldGetLabel(
   ULONG msg                    //32-bit ARINC 429 word
)
```

RETURNS

The 8-bit label field of an ARINC 429 word.

DESCRIPTION

Extracts the 8-bit label field by masking the ARINC 429 word in *msg* 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_FldPutLabel

FldGetParity

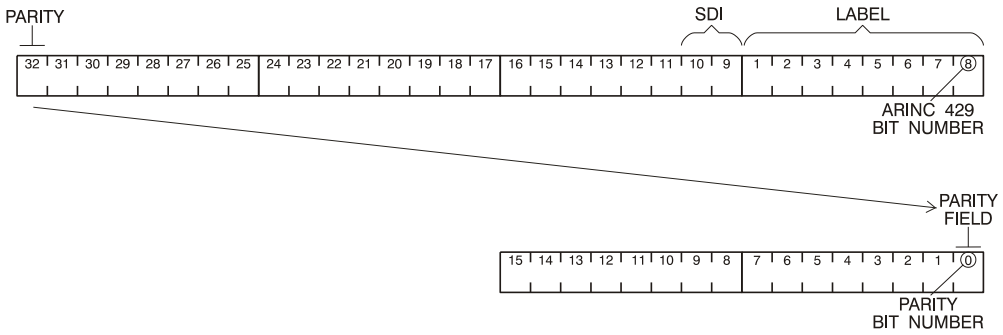
```
USHORT BTI429_FldGetParity(
   ULONG msg                    //32-bit ARINC 429 word
)
```

RETURNS

The parity bit of an ARINC 429 word.

DESCRIPTION

Extracts the parity bit of the ARINC 429 word in *msg*. The extracted parity bit is right-shifted and zero-filled 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_FldGetData

FldGetSDI

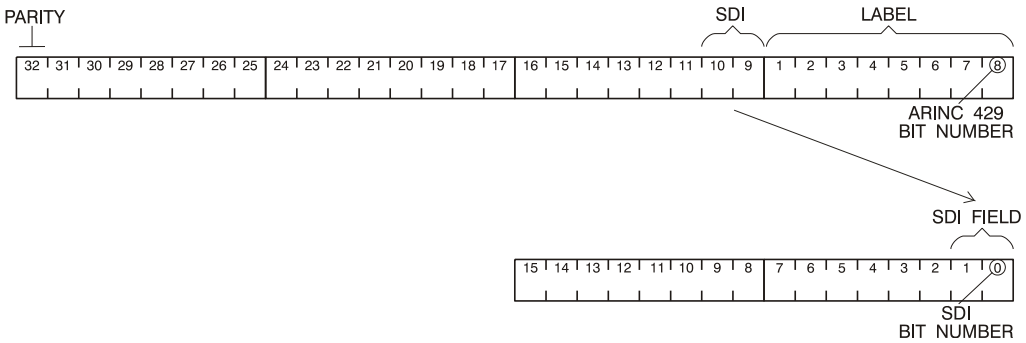
```
USHORT BTI429_FldGetSDI(
   ULONG msg                    //32-bit ARINC 429 word
)
```

RETURNS

The 2-bit SDI field of an ARINC 429 word.

DESCRIPTION

Extracts the 2-bit SDI field of the ARINC 429 word in msg. The extracted SDI field is right-shifted and zero-filled 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_FldPutSDI

FldGetValue

```
ULONG BTI429_FldGetValue(
   ULONG msg,                  //32-bit ARINC 429 word
   USHORT startbit,            //Starting bit number of BCD field
   USHORT endbit               //Ending bit number of BCD field
)
```

RETURNS

The specified field of an ARINC 429 word.

DESCRIPTION

Extracts the value of a specified a bit field from the ARINC 429 word in *msg*. *startbit* (zero-based) and *endbit* (zero-based) determine the lowest and highest bit position of the field to extract. The extracted field is right-shifted and zero-filled.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_FldPutValue

FldPutData

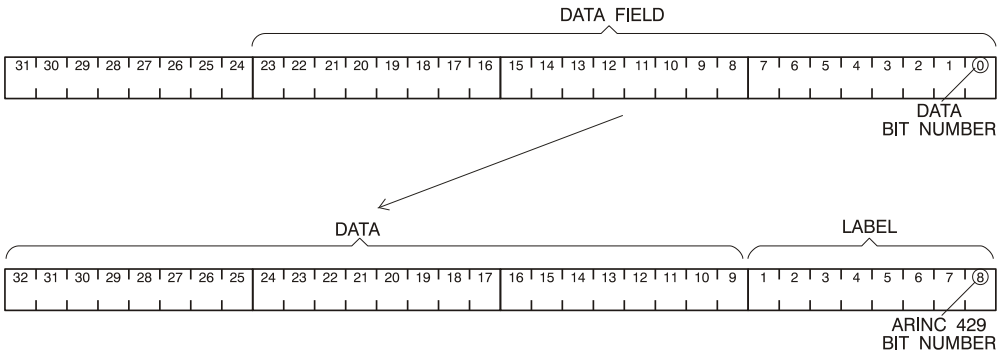
```
ULONG BTI429_FldPutData(
   ULONG msg,                    //32-bit ARINC 429 word
   ULONG data                    //New 23-bit data field value
)
```

RETURNS

The new 32-bit ARINC 429 word with the data field inserted.

DESCRIPTION

Inserts a 24-bit field including both data and parity into the ARINC 429 word in *msg*. *data* is left-shifted and packed into *msg* 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_FldGetData

FldPutLabel

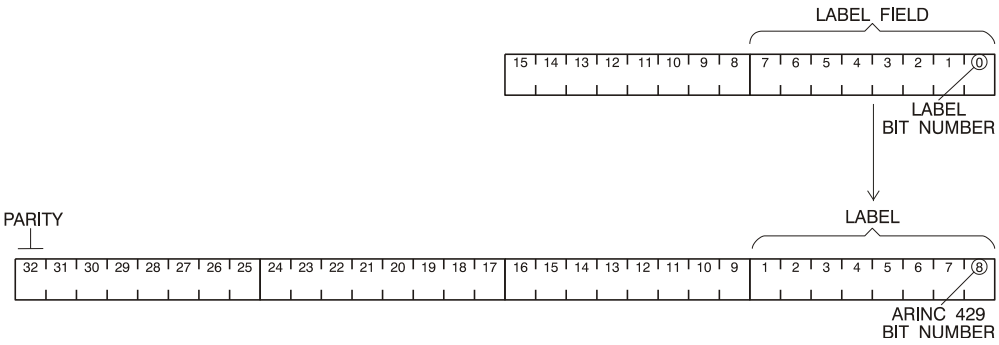
```
ULONG BTI429_FldPutLabel(
   ULONG msg,                    //32-bit ARINC 429 word
   USHORT label                  //New 8-bit label field value
)
```

RETURNS

The new 32-bit ARINC 429 word with the label field inserted.

DESCRIPTION

Inserts an 8-bit label field value into the ARINC 429 word in *msg*. *label* is packed into *msg* 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_FldGetLabel

FldPutSDI

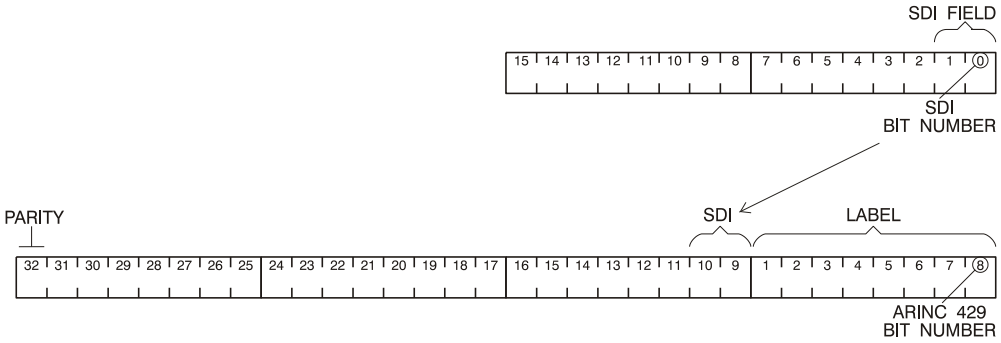
```
ULONG BTI429_FldPutSDI(
   ULONG msg,                  //32-bit ARINC 429 word
   USHORT sdi                  //New 2-bit SDI field value
)
```

RETURNS

The new 32-bit ARINC 429 word with the SDI field inserted.

DESCRIPTION

Inserts a 2-bit SDI field value into the ARINC 429 word in *msg*. *sdi* is left-shifted and packed into *msg* 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_FldGetSDI

FldPutValue

```
ULONG BTI429_FldPutValue(
   ULONG msg,                  //32-bit ARINC 429 word
   ULONG data,                 //New 23-bit data field value
   USHORT startbit,            //Starting bit position of field
   USHORT endbit               //Ending bit position of field
)
```

RETURNS

The new 32-bit ARINC 429 word with the specified field inserted.

DESCRIPTION

Inserts a bit field value into the ARINC 429 word in *msg*. *startbit* (zero-based) and *endbit* (zero-based) specify the low and high bit positions of the field.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_FldGetValue

IntClear

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

RETURNS

None.

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices except those controlled via RPC.

WARNINGS

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

SEE ALSO

BTICard_IntInstall, BTICard_IntUninstall

IntInstall

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

RETURNS

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

DESCRIPTION

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

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

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

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

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

DEVICE DEPENDENCY

Applies to all Devices except those controlled via RPC.

WARNINGS

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

SEE ALSO

BTICard_EventLogRd, BTICard_IntUninstall

IntUninstall

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

RETURNS

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

DESCRIPTION

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

DEVICE DEPENDENCY

Applies to all Devices except those controlled via RPC.

WARNINGS

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

SEE ALSO

BTICard_IntInstall

IRIGConfig

```
ERRVAL BTICard_IRIGConfig(
   ULONG ctrlflags,          //Selects IRIG configuration options
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Configures the on-board IRIG circuit as defined by *ctrlflags* (see table below) for the core specified by *hCore*. IRIG timers are configured and enabled for each core independently.

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
IRIGCFG_DEFAULT	Select all default settings (**bold** below)	✓	✓
IRIGCFG_ENABLE	Enables the IRIG timer	✓	✓
IRIGCFG_DISABLE	Disables the IRIG timer	✓	✓
IRIGCFG_SPEEDB	Enables IRIGB timing	✓	✓
IRIGCFG_SPEEDA	Enables IRIGA timing	✓	✓
IRIGCFG_INTERNAL	Use internal IRIG bus	✓	✓
IRIGCFG_EXTERNAL	Use external IRIG bus	✓	✓
IRIGCFG_SLAVE	IRIG timer for this core is a slave	✓	
IRIGCFG_MASTER	IRIG timer for this core is the master		✓
IRIGCFG_PPS	Enables pulse per second signaling	✓	✓
IRIGCFG_PWM	Enables pulse width modulated signaling	✓	✓
IRIGCFG_AM	Enables amplitude modulated signaling	✓	

DEVICE DEPENDENCY

Applies only to 4G and 5G Devices. Only 5G Devices support amplitude modulated decoding. When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format.

WARNINGS

Rounding is used when the IRIGCFG_PPS option is enabled. Values get rounded up when above 500ms and are rounded down when below 500ms if signaling is configured for pulse per second (PPS).

SEE ALSO

BTICard_IRIGRd, BTICard_IRIGWr, BTICard_IRIGInputThresholdSet, BTICard_IRIGInputThresholdGet, BTICard_TimerStatus

IRIGFieldGet??

```
ULONG BTICard_IRIGFieldGetDays
ULONG BTICard_IRIGFieldGetHours
ULONG BTICard_IRIGFieldGetMin
ULONG BTICard_IRIGFieldGetSec
ULONG BTICard_IRIGFieldGetMillisec
ULONG BTICard_IRIGFieldGetMicrosec(
   ULONG irigvalh,        //Upper 32 bits of the 64-bit BCD IRIG time-tag
   ULONG irigvall,        //Lower 32 bits of the 64-bit BCD IRIG time-tag
)
```

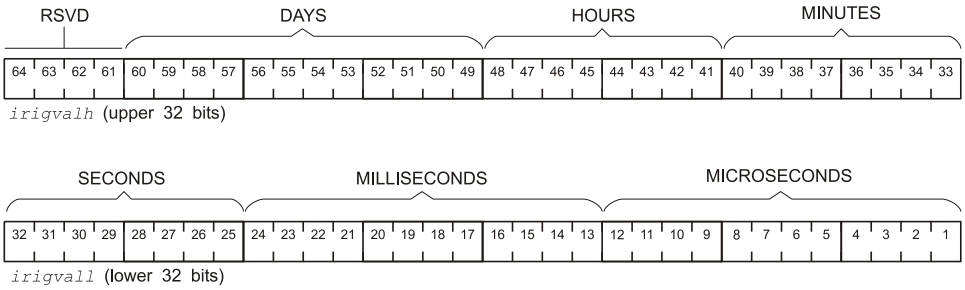
RETURNS

The integer value of the IRIG field for which the function is named.

DESCRIPTION

Extracts the specified BCD field from the 64-bit IRIG time-tag, converts it to an integer, and returns the integer.

An IRIG time-tag is divided into the following BCD fields:



Note: These are utility functions and do not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGFieldPut??

IRIGFieldPut??

```
VOID BTICard_IRIGFieldPutDays
VOID BTICard_IRIGFieldPutHours
VOID BTICard_IRIGFieldPutMin
VOID BTICard_IRIGFieldPutSec
VOID BTICard_IRIGFieldPutMillisec
VOID BTICard_IRIGFieldPutMicrosec(
   ULONG value,          //Field value to write to the BCD IRIG time
   LPULONG irigvalh,     //Pointer to a variable for the upper 32 bits
   LPULONG irigvall,     //Pointer to a variable for the lower 32 bits
)
```

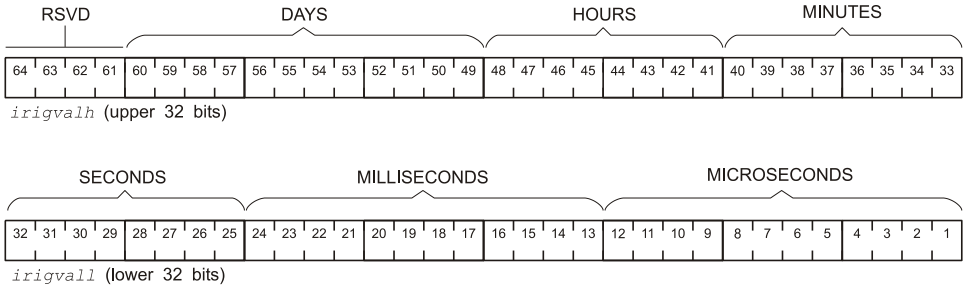
RETURNS

Nothing.

DESCRIPTION

Converts an integer (*value*) to BCD and inserts the BCD value into the specified field in the 64-bit IRIG time-tag.

An IRIG time-tag is divided into the following BCD fields:



Note: These are utility functions and do not access any Device hardware.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGFieldGet??

IRIGInputThresholdGet

```
ERRVAL BTICard_IRIGInputThresholdGet(
   LPUSHORT dacval,          //Pointer to a digital-analog converter value
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Gets the threshold of the IRIG input circuitry. The parameter *dacval* represents a linear voltage scale from 0x0000 to 0xFFF0 see product manual for voltage range. Only the most significant 12 bits are used.

DEVICE DEPENDENCY

Applies to all 5G Devices, except AB1xxx Devices.

WARNINGS

Should only be used when IRIG circuit is configured for Amplitude Modulated (AM) input signaling. Changing the input threshold affects both AM and Pulse Width Modulated decoding.

SEE ALSO

BTICard_IRIGConfig, BTICard_IRIGInputThresholdSet

IRIGInputThresholdSet

```
ERRVAL BTICard_IRIGInputThresholdSet(
   USHORT dacval,          //Digital-analog converter value
   HCORE hCore             //Core handle
)
```

RETURNS

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

DESCRIPTION

Sets the threshold of the IRIG input circuitry. The parameter *dacval* represents a linear voltage scale from 0x0000 to 0xFFF0 see product manual for voltage range. Only the most significant 12 bits are used.

DEVICE DEPENDENCY

Applies to all 5G Devices, except AB1xxx Devices.

WARNINGS

Should only be used when IRIG circuit is configured for Amplitude Modulated (AM) input signaling. Changing the input threshold affects both AM and Pulse Width Modulated decoding.

SEE ALSO

BTICard_IRIGConfig, BTICard_IRIGInputThresholdGet

IRIGRd

```
ERRVAL BTICard_IRIGRd(
   LPBTIIRIGTIME irigtime,  //Pointer to an IRIG time structure
   HCORE hCore              //Core handle
)
```

RETURNS

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

DESCRIPTION

Reads the current value of the IRIG timer from the core specified by *hCore* and puts the value into the *irigtime* structure.

BTIIRIGTIME structure		
Field	**Size**	**Description**
days	USHORT	Day of the year (0–365; January 1st = 0)
hours	USHORT	Hours after midnight (0–23)
min	USHORT	Minutes after the hour (0–59)
sec	USHORT	Seconds after the minute (0–59)
msec	USHORT	Milliseconds after the second (0–999)
usec	USHORT	Microseconds after the millisecond (0–999)

Note: To read the binary timer, see BTICard_TimerRd.

DEVICE DEPENDENCY

Applies only to 4G and 5G Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGWr, BTICard_IRIGConfig, BTICard_TimerRd

IRIGSyncStatus

```
BOOL BTICard_IRIGSyncStatus(
   HCORE hCore                    //Core handle
)
```

RETURNS

TRUE if the IRIG timer is synchronized, or FALSE if it is not synchronized.

DESCRIPTION

Reports the status of the IRIG timer on *hCore* in synchronizing to the signal on the IRIG bus.

DEVICE DEPENDENCY

Applies only to 4G and 5G Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGConfig, BTICard_IRIGWr, BTICard_IRIGRd

IRIGTimeBCDToBin

```
VOID BTICard_IRIGTimeBCDToBin(
   LPULONG timevalh,      //Pointer to upper 32 bits of binary time value
   LPULONG timevall,      //Pointer to lower 32 bits of binary time value
   ULONG irigvalh,        //Upper 32 bits of BCD IRIG time value
   ULONG irigvall,        //Lower 32 bits of BCD IRIG time value
)
```

RETURNS

None.

DESCRIPTION

Converts the 64 bit IRIG BCD time value to the equivalent binary time value.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGTimeBinToBCD

IRIGTimeBinToBCD

```
VOID BTICard_IRIGTimeBinToBCD(
   LPULONG irigvalh,  //Pointer to upper 32 bits of BCD IRIG time value
   LPULONG irigvall,  //Pointer to lower 32 bits of BCD IRIG time value
   ULONG timevalh,    //Upper 32 bits of binary time value
   ULONG timevall,    //Lower 32 bits of binary time value
)
```

RETURNS

None.

DESCRIPTION

Converts the binary time value to the equivalent 64 bit IRIG BCD time value.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGTimeBCDToBin

IRIGWr

```
ERRVAL BTICard_IRIGWr(
   LPBTIIRIGTIME irigtime,  //Pointer to an IRIG time array
   HCORE hCore              //Core handle
)
```

RETURNS

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

DESCRIPTION

Sets the IRIG timer to *irigtime* on the core specified by *hCore*.

IRIGTIME structure		
Field	**Size**	**Description**
days	USHORT	Day of the year (0–365; January 1st = 0)
hours	USHORT	Hours after midnight (0–23)
min	USHORT	Minutes after hour (0–59)
sec	USHORT	Seconds after minute (0–59)
msec	USHORT	Milliseconds after minute (0–999)
usec	USHORT	Microseconds after millisecond (0–999)

Note: To write to the binary timer, see BTICard_TimerWr.

DEVICE DEPENDENCY

Applies only to 4G and 5G Devices.

WARNINGS

None.

SEE ALSO

BTICard_IRIGRd, BTICard_IRIGConfig, BTICard_TimerWr

ListAsyncCreate

```
LISTADDR BTI429_ListAsyncCreate(
   ULONG ctrlflags,        //Selects list options
   INT count,              //One more than the number of entries in list
   INT channel,            //Channel number to associate with List Buffer
   HCORE hCore             //Core handle
)
```

RETURNS

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

DESCRIPTION

Creates an asynchronous transmit List Buffer the size of *count* entries. The List Buffer is connected with the channel specified by *channel*. Every time a gap is encountered in a transmit Schedule, if data exists in the list, it will be transmitted during the gap time. If data does not exist in the list, then the gap time is executed instead (gap times are processed in 36 bit-time increments, which is the size of a word plus a minimum gap). The maximum number of entries that may be stored in the list is *count* - 1.

ctrlflags specifies what type of List Buffer is created and the options associated with the List Buffer. Only FIFO mode is valid for an asynchronous List Buffer. ARINC 429 words are passed to the asynchronous List Buffer with BTI429_ListDataWr and are transmitted only once.

ctrlflags	
Constant	**Description**
LISTCRT429_DEFAULT	Select all default settings (**bold** below)
LISTCRT429_FIFO	Selects FIFO mode
LISTCRT429_PINGPONG	Selects ping-pong mode
LISTCRT429_CIRCULAR	Selects circular mode
LISTCRT429_NOLOG	An entry will not be created in the Event Log List when the List Buffer is empty/full
LISTCRT429_LOG	An entry will be created in the Event Log List when the List Buffer is empty/full

DEVICE DEPENDENCY

3G and 4G Devices support up to 16,376 list entries while 5G Devices support up to 32,764 list entries.

WARNINGS

Normally, this function should be used to associate an asynchronous List Buffer with a transmit channel before the Schedule is created for that channel. Do not use BTI429_MsgDataRd and BTI429_MsgDataWr as they will return incorrect results.

SEE ALSO

BTI429_ListRcvCreate, BTI429_ListXmtCreate, BTI429_ListDataWr, BTI-429_ListDataRd

ListDataBlkRd

```
BOOL BTI429_ListDataBlkRd(
   ULONG data[],              //Pointer to destination buffer
   LPUSHORT datacount,        //Size of destination, number of words read
   LISTADDR list,             //List from which to read data
   HCORE hCore                //Core handle
)
```

RETURNS

A non-zero value if the function succeeded, or zero if unable to read the list.

DESCRIPTION

Reads multiple 32-bit ARINC messages from the *list* and automatically updates the pointers for the next message. It is similar to BTI429_ListDataRd except it reads multiple messages from the list in a single operation. *data* points to the buffer to receive the data words.

When called, *datacount* must point to a variable that contains maximum number of 32-bit ARINC messages that *data* can hold. On return, the variable will contain the number of 32-bit ARINC messages written to *data*.

The *list* must have been returned when the list was created using BTI429_ListRcvCreate.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Function fails if the list is in Ping-Pong mode.

SEE ALSO

BTI429_ListDataBlkWr, BTI429_ListDataRd, BTI429_MsgDataRd, BTI429_ListRcvCreate, BTI429_ListXmtCreate

ListDataBlkWr

```
BOOL BTI429_ListDataBlkWr(
   ULONG data[],              //Pointer to source buffer
   LPUSHORT datacount,        //Size of source, number of words written
   LISTADDR list,             //List to write data to
   HCORE hCore                //Core handle
)
```

RETURNS

A non-zero value if the function succeeded, or zero if unable to wrote to the list.

DESCRIPTION

Writes multiple 32-bit ARINC messages to the *list* and automatically updates the pointers for the next message. It is similar to BTI429_ListDataWr except it writes multiple messages to the *list* in a single operation. *data* points to the buffer containing the data words to write.

When called, *datacount* must point to a variable that contains the total number of 32-bit data words that *data* contains. On return, *datacount* will contain the number of 32-bit ARINC messages that were written to the *list*.

The *list* must have been returned when the *list* was created using BTI429_List-XmtCreate or BTI429_ListAsyncCreate.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Function fails if the list is in Ping-Pong mode.

SEE ALSO

BTI429_ListDataBlkRd, BTI429_ListDataWr, BTI429_MsgDataWr, BTI429_ListRcvCreate, BTI429_ListXmtCreate

ListDataRd

```
ULONG BTI429_ListDataRd(
   LISTADDR list,              //List Buffer from which to read data
   HCORE hCore                 //Core handle
)
```

RETURNS

The 32-bit value of the ARINC word if there is a word in the list, or zero if it is empty.

DESCRIPTION

This function reads one message from the list, and automatically updates the pointers for the next message. It is similar to BTI429_MsgDataRd except it reads from a List Buffer. The *list* parameter is the value returned when the List Buffer was created using BTI429_ListRcvCreate. The position of the message to be read is determined by the mode of the list as follows:

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

list must be configured as a receive List Buffer using BTI429_ListRcvCreate.

SEE ALSO

BTI429_MsgDataRd, BTI429_ListDataWr, BTI429_ListRcvCreate, BTI-429_ListXmtCreate

ListDataWr

```
BOOL BTI429_ListDataWr(
   ULONG value,              //32-bit ARINC word value
   LISTADDR list,            //List to write new data
   HCORE hCore               //Core handle
)
```

RETURNS

TRUE if the operation was successful, or FALSE if an error occurred or the function was unable to write to the list.

DESCRIPTION

This function writes one message to the list and automatically updates the pointers for the next message. It is similar to BTI429_MsgDataWr except it writes the 32-bit ARINC data value specified by *value* to a List Buffer. The *list* parameter is the value returned when the List Buffer was created using BTI-429_ListXmtCreate or BTI429_ListAsyncCreate. The position to which the message is written in the list is determined by the mode of the list as follows:

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

list must be configured as a write transmit List Buffer using either BTI429_-ListXmtCreate or BTI429_ListAsyncCreate

SEE ALSO

BTI429_MsgDataWr, BTI429_ListDataRd, BTI429_ListRcvCreate, BTI-429_ListXmtCreate, BTI429_ListAsyncCreate

ListRcvCreate

```
LISTADDR BTI429_ListRcvCreate(
   ULONG ctrlflags,        //Selects list options
   INT count,              //One more than the number of entries in list
   MSGADDR message,        //Message address to associate with List Buffer
   HCORE hCore             //Core handle
)
```

RETURNS

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

DESCRIPTION

Creates a receive List Buffer the size of *count* entries. The List Buffer is connected with a Message Record so that the data is processed in the list instead of in the Message Record. The *ctrlflags* specify what type of List Buffer is created and the options associated with the List Buffer. Only FIFO and ping-pong modes are applicable to a receive List Buffer. The maximum number of entries that may be stored in the list is *count* - 1.

ctrlflags	
Constant	**Description**
LISTCRT429_DEFAULT	Select all default settings (**bold** below)
LISTCRT429_FIFO	Selects FIFO mode
LISTCRT429_PINGPONG	Selects ping-pong mode
LISTCRT429_CIRCULAR	Selects circular mode
LISTCRT429_NOLOG	An entry will not be created in the Event Log List when the List Buffer is empty/full
LISTCRT429_LOG	An entry will be created in the Event Log List when the List Buffer is empty/full

DEVICE DEPENDENCY

LISTCRT429_PINGPONG is not supported by 5G Devices since they inherently have protection for data coherency (use BTI429_MsgCommRd and BTI429_MsgCommWr instead).

3G and 4G Devices support up to 16,376 list entries while 5G Devices support up to 32,764 list entries.

WARNINGS

After connecting *message* with a List Buffer, the functions BTI429_ListDataRd and BTI429_ListDataWr must be used to read or write the data. Do not use BTI429_MsgDataRd and BTI429_MsgDataWr as they will return incorrect results.

SEE ALSO

BTI429_ListXmtCreate, BTI429_ListAsyncCreate, BTI429_ListDataWr, BTI429_ListDataRd

ListStatus

```
INT BTI429_ListStatus(
   LISTADDR list,              //Address of the List Buffer
   HCORE hCore                 //Core handle
)
```

RETURNS

The status value of the specified List Buffer.

DESCRIPTION

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

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_ListAsyncCreate, BTI429_ListDataRd, BTI429_ListDataWr, BTI-429_ListRcvCreate, BTI429_ListXmtCreate

ListXmtCreate

```
LISTADDR BTI429_ListXmtCreate(
   ULONG ctrlflags,        //Selects list options
   INT count,              //One more than the number of entries in list
   MSGADDR message,        //Message address to associate with List Buffer
   HCORE hCore             //Core handle
)
```

RETURNS

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

DESCRIPTION

Creates a transmit List Buffer the size of *count* entries. The List Buffer is connected with a Message Record so that the data is processed in the list instead of in the Message Record. The *ctrlflags* specify what type of List Buffer is created and the options associated with the List Buffer. The maximum number of entries that may be stored in the list is *count* - 1.

ctrlflags	
Constant	**Description**
LISTCRT429_DEFAULT	Select all default settings (**bold** below)
LISTCRT429_FIFO	Selects FIFO mode
LISTCRT429_PINGPONG	Selects ping-pong mode
LISTCRT429_CIRCULAR	Selects circular mode
LISTCRT429_NOLOG	An entry will not be created in the Event Log List when the List Buffer is empty/full
LISTCRT429_LOG	An entry will be created in the Event Log List when the List Buffer is empty/full

DEVICE DEPENDENCY

LISTCRT429_PINGPONG is not supported by 5G Devices since they inherently have protection for data coherency (use BTI429_MsgCommRd and BTI429_MsgCommWr instead).

3G and 4G Devices support up to 16,376 list entries while 5G Devices support up to 32,764 list entries.

WARNINGS

After connecting *message* with a List Buffer, the functions BTI429_ListDataRd and BTI429_ListDataWr must be used to read or write the data. Do not use BTI429_MsgDataRd and BTI429_MsgDataWr as they will return incorrect results.

SEE ALSO

BTI429_ListRcvCreate, BTI429_ListAsyncCreate, BTI429_ListDataWr, BTI429_ListDataRd

MsgBlockRd

```
MSGADDR BTI429_MsgBlockRd(
   LPMSGFIELDS429 msgfields, //Pointer to destination structure
   MSGADDR message,          //Message from which to read
   HCORE hCore               //Core handle
)
```

RETURNS

The address of the message structure that was read.

DESCRIPTION

Reads an entire Message Record from the Device.

MSGFIELDS429 structure		
Field	**Size**	**Description**
msgopt	USHORT	Message options fields. Do not modify these fields.
msgact	USHORT	Message activity. See table below for detail.
msgdata	ULONG	Message data value. A 32-bit ARINC 429 data word value.
listptr	ULONG	List Buffer pointer. Used instead of msgdata when in List Buffer mode.
timetag	ULONG	Lower 32 bits of the time-tag value.
hitcount	ULONG	Hit Counter value. Used instead of time-tag when in Hit Counter mode.
maxtime	ULONG	Maximum repetition rate. 32 bits with resolution equal to time-tag resolution.
elapsetime	ULONG	Elapsed time. 32 bits with resolution equal to time-tag resolution. Used instead of maxtime when in Elapsed Time mode.
mintime	ULONG	Minimum repetition rate. 32 bits with resolution equal to time-tag resolution.
userptr	ULONG	Reserved
timetagh	ULONG	Upper 32 bits of the time-tag value
decgap	USHORT	Measured decoder gap time (4 bits)
rsvd	USHORT	Reserved

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

msgact field	
Constant	**Description**
MSGACT429_CHMASK	The channel number mask value. Shift the result right with MSGACT429_CHSHIFT.
MSGACT429_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACT429_SPD	This bit reflects the speed detected. A one signifies high speed (100 Kbps), and a zero signifies low speed (12.5 Kbps).
MSGACT429_ERR	If set, it signifies that an error occurred in receiving this word. The type of error is defined by the following bits.
MSGACT429_GAP	Gap Error. A gap of less than four bit times preceded the word.
MSGACT429_PAR	Parity error. A parity error was detected in the word.
MSGACT429_LONG	Long word error. A word of more than 32 bits was detected.
MSGACT429_BIT	Bit timing error. An error occurred while decoding the bits of the word (short bits or long bits).
MSGACT429_TO	Time out error. The decoder timed out while receiving a word (short word).
MSGACT429_HIT	Signifies that the message has been processed by the firmware (the Hit bit).

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

None.

SEE ALSO

BTI429_MsgBlockWr, BTI429_MsgCommRd, BTI429_MsgDataRd, BTI429_MsgDataWr, BTI429_FilterSet, BTI429_FilterDefault

MsgBlockWr

```
MSGADDR BTI429_MsgBlockWr(
   LPMSGFIELDS429 msgfields,//Pointer to source structure
   MSGADDR message,         //Message to write to
   HCORE hCore              //Core handle
)
```

RETURNS

The address of the message structure that was written.

DESCRIPTION

Writes an entire Message Record to the Device. This function is used to modify certain fields in a Message Record after the Message Record was read using BTI429_MsgBlockRd. Only the msgdata, hitcount, mintime, and maxtime fields should be modified. All other fields should be restored to the value read. See BTI429_MsgBlockRd for a list of the Message Record fields.

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Do not modify any fields other than the four listed above.

SEE ALSO

BTI429_MsgBlockRd, BTI429_MsgCommWr, BTI429_MsgDataRd, BTI429_MsgDataWr, BTI429_MsgCreate, BTI429_FilterSet, BTI429_FilterDefault

MsgCommRd

```
MSGADDR BTI429_MsgCommRd(
   LPMSGFIELDS429 msgfields, //Pointer to destination structure
   MSGADDR message,          //Message from which to read
   HCORE hCore               //Core handle
)
```

RETURNS

The address of the message structure that was read.

DESCRIPTION

Reads an entire message structure from the core. Similar to BTI429_MsgBlockRd, except it uses non-contended accesses of Device memory. See BTI429_MsgBlockRd for a list of the Message Record fields.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgBlockRd, BTI429_MsgCommWr, BTI429_MsgDataRd, BTI429_-MsgDataWr, BTI429_FilterSet, BTI429_FilterDefault

MsgCommWr

```
MSGADDR BTI429_MsgCommWr(
   LPMSGFIELDS429 msgfields,//Pointer to source structure
   MSGADDR message,         //Message to write to
   HCORE hCore              //Core handle
)
```

RETURNS

The address of the message structure that was written.

DESCRIPTION

Writes an entire message structure to the core. Similar to BTI429_MsgBlkWr, except it uses non-contended accesses of Device memory. This function is used to modify certain fields in a Message Record after the Message Record was read using BTI429_MsgBlockRd. Only the msgdata, hitcount, mintime, and maxtime fields should be modified. All other fields should be restored to the value read. See BTI429_MsgBlockRd for a list of the Message Record fields.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Do not modify any fields other than the four listed above.

SEE ALSO

BTI429_MsgCommRd, BTI429_MsgBlockWr, BTI429_MsgDataWr, BTI-429_MsgCreate, BTI429_FilterSet, BTI429_FilterDefault

MsgCreate

```
MSGADDR BTI429_MsgCreate(
   ULONG ctrlflags,          //Selects message options
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

ctrlflags			
Constant	**Description**	**Rcv**	**Xmt**
MSGCRT429_DEFAULT	Select all default settings (**bold** below)	✓	✓
MSGCRT429_NOSEQ	This message will not get recorded in the Sequential Record	✓	✓
MSGCRT429_SEQ	This message will get recorded in the Sequential Record	✓	✓
MSGCRT429_NOLOG	This message will not create an entry in the Event Log List	✓	✓
MSGCRT429_LOG	This message will create an entry in the Event Log List	✓	✓
MSGCRT429_NOTIMETAG	This message will not record a time-tag	✓	✓
MSGCRT429_TIMETAG	This message will record a time-tag	✓	✓
MSGCRT429_NOELAPSE	This message will not record an Elapsed Time	✓	✓
MSGCRT429_ELAPSE	This message will record an Elapsed Time	✓	✓
MSGCRT429_NOMAXMIN	This message will not record maximum and minimum repetition rates	✓	✓
MSGCRT429_MAX	This message will record maximum repetition rates	✓	✓
MSGCRT429_MIN	This message will record minimum repetition rates	✓	✓
MSGCRT429_MAXMIN	This message will record maximum and minimum repetition rates	✓	✓
MSGCRT429_NOHIT	This message will not record a Hit Counter	✓	✓
MSGCRT429_HIT	This message will record a Hit Counter (can't have Hit Counter with time-tag, elapse, or max/min timing)	✓	✓
MSGCRT429_NOSKIP	This message will not be skipped	✓	✓
MSGCRT429_SKIP	This message will be skipped, and none of the options will be processed	✓	✓
MSGCRT429_NOSYNC	This message will not generate a SYNCOUT signal		✓
MSGCRT429_SYNC	This message will generate a SYNCOUT signal		✓
MSGCRT429_NOEXTRIG	This message will be triggered immediately		✓
MSGCRT429_EXTRIG	This message will wait for an EXTRIG* pulse to be triggered		✓
MSGCRT429_NOERR	This message will not have a parity error		✓
MSGCRT429_PARERR	This message will have a parity error		✓
MSGCRT429_WIPE	This data fields of this message will initially be wiped to a value	✓	✓
MSGCRT429_NOWIPE	The data fields of this message will initially be left unchanged	✓	✓
MSGCRT429_WIPE0	The data fields of this message will be wiped with a value of zeros (this option does not get used if MSGCRT429_NOWIPE is used)	✓	✓
MSGCRT429_WIPE1	The data fields of this message will be wiped with a value of ones (this option does not get used if MSGCRT429_NOWIPE is used)	✓	✓

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgDataRd, BTI429_MsgDataWr, BTI429_MsgBlockRd, BTI429_-MsgBlockWr , BTI429_MsgCommRd, , BTI429_MsgCommWr

MsgDataRd

```
ULONG BTI429_MsgDataRd(
   MSGADDR message,          //Message from which to read
   HCORE hCore               //Core handle
)
```

RETURNS

32-bit value of the ARINC data word.

DESCRIPTION

Reads the 32-bit value of the ARINC data word from the Message Record specified by *message*.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgDataWr, BTI429_MsgCreate, BTI429_MsgBlockRd, BTI429_-MsgCommRd, BTI429_FilterSet, BTI429_FilterDefault

MsgDataWr

```
VOID BTI429_MsgDataWr(
   ULONG value,              //Value of data to write to message
   MSGADDR message,          //Message to receive new data
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Writes the 32-bit ARINC data value specified by value into the Message Record specified by *message*.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgDataRd, BTI429_MsgCreate, BTI429_MsgBlockRd, BTI429_MsgCommWr, BTI429_MsgBlockWr

MsgGroupBlockRd

```
VOID BTI429_MsgGroupBlockRd(
   MSGFIELDS429 msgfields[],     //Array of destination structures
   MSGADDR msgs[],               //Array of message addresses
   INT nummsgs,                  //Number of  messages to read
   HCORE hCore                   //Core handle
)
```

RETURNS

None.

DESCRIPTION

Reads a series of Message Records from the Device. *messages* points to an array of message addresses. The contents of each message in *messages* is written to its respective entry in *msgfields*. There must be *numsgs* entries in both *messages* and *msgfields*. See BTI429_MsgBlockRd for a list of the Message Record fields.

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

None.

SEE ALSO

BTI429_MsgBlockWr, BTI429_MsgCommRd, BTI429_MsgDataRd, BTI429_MsgGroupBlockWr, BTI429_MsgDataWr, BTI429_FilterSet, BTI429_FilterDefault

MsgGroupBlockWr

```
VOID BTI429_MsgGroupBlockWr(
   LPMSGFIELDS429 msgfields,     //Pointer to source structures
   MSGADDR msgs[],               //Messages to write to
   INT nummsgs,                  //Number of messages to write
   HCORE hCore                   //Core handle
)
```

RETURNS

None.

DESCRIPTION

Writes a series of Message Records to the Device. *messages* points to an array of message addresses. Each entry in *msgfields* is written to its respective message in *messages*. There must be *numsgs* entries in both *msgs* and *msgfields*.

This function is used to modify certain fields in Message Records after the Message Records were read using BTI429_MsgGroupBlockRd or BTI429_MsgBlockRd. Only the msgdata, hitcount, mintime, and maxtime fields should be modified. All other fields should be restored to the value read. See BTI429_MsgBlockRd for a list of the Message Record fields.

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Do not modify any fields other than the four listed above.

SEE ALSO

BTI429_MsgGroupBlockRd, BTI429_MsgBlockRd, BTI429_MsgDataRd, BTI429_MsgDataWr, BTI429_MsgCreate, BTI429_FilterSet, BTI429_FilterDefault

MsgGroupRd

```
VOID BTI429_MsgGroupRd(
   ULONG msgdata[],          //Pointer to destination array
   MSGADDR msgs[],           //Array of Messages to read from
   INT nummsgs,              //Number of messages to read
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Reads the 32-bit ARINC data word from multiple Message Records in a single operation. It is similar to BTI429_MsgDataRd except it reads *nummsgs* Message Records instead of just one Message Record. The array *msgs* points to the message addresses to read from. The array *msgdata* points to the memory to receive the 32-bit ARINC data word of each message.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgGroupWr, BTI429_MsgDataRd

MsgGroupWr

```
VOID BTI429_MsgGroupWr(
   ULONG msgdata[],          //Pointer to source array
   MSGADDR msgs[],           //Array of Messages to write to
   INT nummsgs,              //Number of messages to write
   HCORE hCore               //Core handle
)
```

RETURNS

None.

DESCRIPTION

Writes the 32-bit ARINC data word to multiple Message Records in a single operation. It is similar to BTI429_MsgDataWr except it writes *nummsgs* Message Records instead of just one Message Record. The array *msgdata* points to the 32-bit ARINC data words to write. The array *msgs* points to message addresses to write to.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgGroupRd, BTI429_MsgDataWr

MsgIsAccessed

```
BOOL BTI429_MsgIsAccessed(
   MSGADDR message,          //Message to test if it has been accessed
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

This function indicates that a Message Record has been processed by either transmission or reception of a message. BTI429_MsgIsAccessed returns the value of the Hit bit, then clears it.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgDataRd, BTI429_MsgCreate, BTI429_MsgBlockRd, BTI-429_MsgCommRd, BTI429_MsgBlockWr, BTI429_FilterSet, BTI429_Filter-Default

MsgMultiSkipWr

```
VOID BTI429_MsgMultiSkipWr(
   BOOL skip[],              //Array of skip values to write
   MSGADDR message[],        //Array of message addresses
   INT count,                //Number of items to update
   HCORE hCore               //Core handle
)
```

RETURNS

TRUE if all messages were successfully updated, otherwise FALSE.

DESCRIPTION

Writes the state of the skip bit for each of the Message Records specified by *message*. If the skip bit is FALSE (zero), the message will be processed as normal. If the bit is TRUE (non-zero), the message will be skipped and will not be processed.

The skip state of the nth element of the *message* array is set by the nth element of the *skip* array.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgSkipRd, BTI429_MsgSkipWr

MsgSkipRd

```
BOOL BTI429_MsgSkipRd(
   MSGADDR message,          //Message to read from
   HCORE hCore               //Core handle
)
```

RETURNS

The state of the skip bit for a message.

DESCRIPTION

Reads the state of the skip bit for the Message Record specified by *message*. If the bit is zero, the message will be processed as normal. If the bit is non-zero, the message will be skipped and will not be processed.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgSkipRd, BTI429_MsgMultiSkipWr

MsgSkipWr

```
VOID BTI429_MsgSkipWr(
   BOOL skip,                  //Skip value to write
   MSGADDR message,            //Message to write to
   HCORE hCore                 //Core handle
)
```

RETURNS

None.

DESCRIPTION

Writes the state of the skip bit for the Message Record specified by *message*. If the bit is zero, the message will be processed as normal. If the bit is non-zero, the message will be skipped and will not be processed.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTI429_MsgSkipRd, BTI429_MsgMultiSkipWr

MsgSyncDefine

```
ERRVAL BTI429_MsgSyncDefine(
   BOOL enable,              //Enable/disable external sync pulse
   USHORT syncmask,          //Line(s) used for sync output
   USHORT pinpolarity,       //Active pin polarity (high/low)
   MSGADDR message,          //Transmit message to drive sync pulse
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Defines the sync output conditions for a transmit message specified by *message* and configures it to output a sync signal. This sync signal appears on the line(s) specified by *syncmask* with the polarity specified by *pinpolarity* (see tables below). BTI429_MsgSyncDefine may be called during run time to redefine the sync output settings.

The constants in the tables below may be bitwise OR-ed together to configure multiple lines.

syncmask	
Constant	**Description**
SYNCMASK_SYNCA	Selects sync line A
SYNCMASK_SYNCB	Selects sync line B
SYNCMASK_SYNCC	Selects sync line C

pinpolarity	
Constant	**Description**
SYNCPOL_SYNCAL	Sets active low polarity for sync line A
SYNCPOL_SYNCAH	Sets active high polarity for sync line A
SYNCPOL_SYNCBL	Sets active low polarity for sync line B
SYNCPOL_SYNCBH	Sets active high polarity for sync line B
SYNCPOL_SYNCCL	Sets active low polarity for sync line C
SYNCPOL_SYNCCH	Sets active high polarity for sync line C

Alternatively, to configure all messages on a specified channel to output a sync pulse, use BTI429_ChSyncDefine.

DEVICE DEPENDENCY

Applies to 4G and 5G Devices. 3G Devices, which have only a single sync line, can call BTI429_MsgCreate with the MSGCRT429_SYNC flag. The mapping of sync lines to discretes is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

This function only applies to transmit messages.

SEE ALSO

BTI429_ChSyncDefine

MsgTriggerDefine

```
ERRVAL BTI429_MsgTriggerDefine(
   BOOL enable,              //Enable/disable external trigger
   USHORT trigmask,          //Line(s) used for trigger signal
   USHORT trigval,           //Active/inactive condition for trigger line(s)
   USHORT pinpolarity,       //Active pin polarity (high/low)
   MSGADDR message,          //Message address
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Defines the trigger input settings for the message specified by *message* and associates it with a trigger signal. The input line(s) are specified by *trigmask* with an active trigger state being the combination of *trigval* and *pinpolarity* (see tables below). When tagged and encountered in a transmit schedule, the *message* waits for the defined trigger condition before transmitting. BTI429_MsgTriggerDefine may be called during run time to redefine the trigger input settings.

The constants in the tables below may be bitwise OR-ed together to configure multiple lines. All combined states must be true for the trigger to occur.

trigmask	
Constant	**Description**
TRIGMASK_TRIGA	Selects trigger line A
TRIGMASK_TRIGB	Selects trigger line B
TRIGMASK_TRIGC	Selects trigger line C

trigval	
Constant	**Description**
TRIGVAL_TRIGAOFF	Trigger on line A inactive
TRIGVAL_TRIGAON	Trigger on line A active
TRIGVAL_TRIGBOFF	Trigger on line B inactive
TRIGVAL_TRIGBON	Trigger on line B active
TRIGVAL_TRIGCOFF	Trigger on line C inactive
TRIGVAL_TRIGCON	Trigger on line C active

pinpolarity	
Constant	**Description**
TRIGPOL_TRIGAL	Sets active low polarity for trigger line A
TRIGPOL_TRIGAH	Sets active high polarity for trigger line A
TRIGPOL_TRIGBL	Sets active low polarity for trigger line B
TRIGPOL_TRIGBH	Sets active high polarity for trigger line B
TRIGPOL_TRIGCL	Sets active low polarity for trigger line C
TRIGPOL_TRIGCH	Sets active high polarity for trigger line C

Alternatively, to associate all messages on a transmit channel with a trigger signal, use BTI429_ChTriggerDefine.

DEVICE DEPENDENCY

Applies to 4G and 5G Devices. 3G Devices, which have a single external trigger can call BTI429_MsgCreate with the MSGCRT429_EXTRIG flag. The mapping of trigger lines to discretes is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTI429_ChTriggerDefine, BTICard_CardTrigger, BTICard_CardTriggerEx

ParamAmplitudeConfig

```
ERRVAL BTI429_ParamAmplitudeConfig(
   ULONG configval,          //Configuration options to set
   USHORT dacval,            //12-bit digital-analog converter value
   INT xmtchan,              //Transmit channel number
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

configval	
Constant	**Description**
PARAMCFG429_DEFAULT	Select all default settings (**bold** below)
PARAMCFG429_AMPLON	Enables parametric amplitude control
PARAMCFG429_AMPLOFF	Disables parametric amplitude control

DEVICE DEPENDENCY

This function may only be used with a 429 transmit channel with parametric capability. A channel can be tested for parametric capability using BTI429_ChGetInfo.

WARNINGS

None.

SEE ALSO

BTI429_ParamBitRateConfig, BTI429_ChGetInfo

ParamBitRateConfig

```
ERRVAL BTI429_ParamBitRateConfig(
   ULONG configval,          //Configuration options to set
   USHORT divval,            //Divide value
   INT channel,              //Channel number
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Configures *channel* for non-standard bus frequency as defined by *configval* (see table below) and sets the clock divider value to *divval*. The *divval* has a maximum value of 255. If parametric bit rate control is not set or turned off, then the frequency reverts to the default value.

configval	
Constant	**Description**
PARAMCFG429_DEFAULT	Select all default settings (**bold** below)
PARAMCFG429_BITRATEON	Enables parametric bit rate control
PARAMCFG429_BITRATEOFF	Disables parametric bit rate control

DEVICE DEPENDENCY

Applies to all 5G Devices. For 3G or 4G, this function may only be used on a 429 channel with parametric capability. A 3G or 4G channel can be tested for bit rate parametric capability using BTI429_ChGetInfo.

Below are the formulas for rate calculations using *divval*.

Formulas for Rate Calculations in kHz		
Generation/Type	**High Speed**	**Low Speed**
4G Encoder and Decoder	4000 / (*divval* + 1)	500 / (*divval* + 1)
5G Decoder	200 / (*divval* + 1)	25 / (*divval* + 1)
5G Encoder	1000 / (*divval* + 1)	125 / (*divval* + 1)

WARNINGS

None.

SEE ALSO

BTI429_ParamAmplitudeConfig, BTI429_ChGetInfo

SchedBranch

```
SCHNDX BTI429_SchedBranch(
   ULONG condition,          //Condition for branch
   SCHNDX destindex,         //Destination index for branch
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

The *condition* flags listed below may be used to specify the branch condition.

condition	
Constant	**Description**
COND429_ALWAYS	Always branch
COND429_DIO1ACT	Branch if DIO1 is active
COND429_DIO1NACT	Branch if DIO1 is inactive
COND429_DIO2ACT	Branch if DIO2 is active
COND429_DIO2NACT	Branch if DIO2 is inactive
COND429_DIO3ACT	Branch if DIO3 is active
COND429_DIO3NACT	Branch if DIO3 is inactive
COND429_DIO4ACT	Branch if DIO4 is active
COND429_DIO4NACT	Branch if DIO4 is inactive
COND429_DIO5ACT	Branch if DIO5 is active
COND429_DIO5NACT	Branch if DIO5 is inactive
COND429_DIO6ACT	Branch if DIO6 is active
COND429_DIO6NACT	Branch if DIO6 is inactive
COND429_DIO7ACT	Branch if DIO7 is active
COND429_DIO7NACT	Branch if DIO7 is inactive
COND429_DIO8ACT	Branch if DIO8 is active
COND429_DIO8NACT	Branch if DIO8 is inactive

DEVICE DEPENDENCY

Does not apply to 5G Devices. The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

A call to BTI429_ChConfig must precede this function. When creating subroutines, BTI429_SchedEntry needs to be called to point to the main section. The Command Block pointed to by *destindex* must have been previously inserted in the Schedule.

SEE ALSO

BTI429_SchedEntry, BTI429_SchedCall, BTI429_SchedReturn

SchedBuild

```
ERRVAL BTI429_SchedBuild(
   INT nummsgs,              //Number of messages to Schedule
   MSGADDR msgs[],           //Array of message addresses
   INT min[],                //Array of message frequencies
   INT max[],                //Array of message frequencies
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Clears any transmit Schedule for the specified channel and creates a new transmit Schedule that sequences messages at given intervals. The new Schedule will consist of *nummsgs* messages, each transmitted within an interval specified by the *min* and *max* interval arrays (in units of milliseconds). *msgs* points to an array of message addresses, each previously generated by a call to BTI429_MsgCreate.

The nth element of the *msgs* array uses the nth element of the *min* and *max* intervals arrays to create the Schedule.

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

If the range of every interval is zero (the min and max value of each pair are the same), then no range checking is performed on the resulting schedule.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function as well as a call to BTI429_MsgCreate for each message to be Scheduled.

SEE ALSO

BTI429_SchedMsg, BTI429_SchedMsgEx, BTI429_SchedGap

SchedCall

```
SCHNDX BTI429_SchedCall(
   ULONG condition,          //Condition for call
   SCHNDX destindex,         //Destination index of subroutine
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

The *condition* flags listed below may be used to specify the call condition.

condition	
Constant	**Description**
COND429_ALWAYS	Always branch
COND429_DIO1ACT	Branch if DIO1 is active
COND429_DIO1NACT	Branch if DIO1 is inactive
COND429_DIO2ACT	Branch if DIO2 is active
COND429_DIO2NACT	Branch if DIO2 is inactive
COND429_DIO3ACT	Branch if DIO3 is active
COND429_DIO3NACT	Branch if DIO3 is inactive
COND429_DIO4ACT	Branch if DIO4 is active
COND429_DIO4NACT	Branch if DIO4 is inactive
COND429_DIO5ACT	Branch if DIO5 is active
COND429_DIO5NACT	Branch if DIO5 is inactive
COND429_DIO6ACT	Branch if DIO6 is active
COND429_DIO6NACT	Branch if DIO6 is inactive
COND429_DIO7ACT	Branch if DIO7 is active
COND429_DIO7NACT	Branch if DIO7 is inactive
COND429_DIO8ACT	Branch if DIO8 is active
COND429_DIO8NACT	Branch if DIO8 is inactive

DEVICE DEPENDENCY

Does not apply to 5G Devices. The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

A call to BTI429_ChConfig must precede this function. After creating subroutines, BTI429_SchedEntry needs to be called to point to the main section. Every use of BTI429_SchedCall must have a corresponding BTI429_SchedReturn. The Command Block pointed to by *destindex* must have been previously inserted in the Schedule.

SEE ALSO

BTI429_SchedEntry, BTI429_SchedBranch, BTI429_SchedReturn

SchedEntry

```
SCHNDX BTI429_SchedEntry(
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

To create a Schedule with subroutines, first define the subroutines by calling the desired Schedule functions while saving the returned Schedule indices. Then call BTI429_SchedEntry to set the starting point of the Schedule. Then build the main part of the Schedule by calling the other Schedule functions that include the commands that call the subroutines (i.e., BTI429_SchedCall and BTI429_SchedBranch).

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTI429_SchedBranch, BTI429_SchedCall, BTI429_SchedReturn

SchedGap

```
SCHNDX BTI429_SchedGap(
   USHORT gap,                  //Gap value in bit times
   INT channel,                 //Channel number of transmitter
   HCORE hCore                  //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a GAP Command Block to the current end of the Schedule. When a GAP Command Block is encountered in the Schedule, it triggers the transmission of any preceding MESSAGE command Block as well as the specified gap (in bit times) before the next message can be transmitted.

Depending on the configuration of an asynchronous List Buffer, this function will internally call either BTI429_SchedGapFixed or BTI429_SchedGapList. This allows the user to include or leave out an asynchronous List Buffer without rebuilding the Schedule.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTI429_SchedMsg, BTI429_SchedGapFixed, BTI429_SchedGapList

SchedGapFixed

```
SCHNDX BTI429_SchedGapFixed(
   USHORT gap,                  //Gap value in bit times
   INT channel,                 //Channel number of transmitter
   HCORE hCore                  //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a FIXEDGAP Command Block to the current end of the Schedule. When a FIXEDGAP Command Block is encountered in the Schedule, it triggers the transmission of any preceding MESSAGE command Block as well as the specified gap (in bit times) before the next message can be transmitted. The gap time is fixed so no asynchronous messages can be transmitted during this time.

Note: This is an advanced function, and for most applications, the BTI429_-SchedGap function is preferred.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTI429_SchedGap, BTI429_SchedGapList

SchedGapList

```
SCHNDX BTI429_SchedGapList(
   USHORT gap,              //Gap value in bit times
   LISTADDR list,           //Address of asynchronous List Buffer
   INT channel,             //Channel number of transmitter
   HCORE hCore              //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a conditional LISTGAP Command Block to the current end of the Schedule. A conditional LISTGAP Command Block specifies *gap* (in bit times) before the next Scheduled message can be transmitted. During this gap time, if a message exists in the asynchronous List Buffer, it is transmitted in the gap time.

Note: This is an advanced function, and for most applications, the BTI429_-SchedGap function is preferred.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function. In addition, an asynchronous List Buffer must be configured using BTI429_ListAsyncCreate before calling this function.

SEE ALSO

BTI429_SchedGap, BTI429_SchedGapFixed, BTI429_ListAsyncCreate

SchedHalt

```
SCHNDX BTI429_SchedHalt(
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a HALT Command Block to the current end of the Schedule. A HALT Command Block stops the Schedule until the channel is re-enabled using BTI429_ChStart. When a HALT Command Block is encountered, it has the same effect as executing the BTI429_ChStop function.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTI429_SchedPause, BTI429_ChStart, BTI429_ChStop

SchedLog

```
SCHNDX BTI429_SchedLog(
   ULONG condition,          //Value to test
   USHORT tagval,            //Event tag value
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a conditional LOG Command Block to the current end of the Schedule. A conditional LOG Command Block causes the Device to generate an Event Log List entry if *condition* evaluates as TRUE. The event type placed in the Event Log List is EVENTTYPE_429OPCODE and the user-specified value *tagval* is used as the info value. Entries are read out of the Event Log List using BTICard_EventLogRd.

The *condition* flags listed below may be used to specify the Event condition.

condition	
Constant	**Description**
COND429_ALWAYS	Always branch
COND429_DIO1ACT	Branch if DIO1 is active
COND429_DIO1NACT	Branch if DIO1 is inactive
COND429_DIO2ACT	Branch if DIO2 is active
COND429_DIO2NACT	Branch if DIO2 is inactive
COND429_DIO3ACT	Branch if DIO3 is active
COND429_DIO3NACT	Branch if DIO3 is inactive
COND429_DIO4ACT	Branch if DIO4 is active
COND429_DIO4NACT	Branch if DIO4 is inactive
COND429_DIO5ACT	Branch if DIO5 is active
COND429_DIO5NACT	Branch if DIO5 is inactive
COND429_DIO6ACT	Branch if DIO6 is active
COND429_DIO6NACT	Branch if DIO6 is inactive
COND429_DIO7ACT	Branch if DIO7 is active
COND429_DIO7NACT	Branch if DIO7 is inactive
COND429_DIO8ACT	Branch if DIO8 is active
COND429_DIO8NACT	Branch if DIO8 is inactive

DEVICE DEPENDENCY

5G Devices only support the COND429_ALWAYS flag. The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTICard_EventLogRd

SchedMode

```
ERRVAL BTI429_SchedMode(
   ULONG modeval              //Schedule Build mode options
)
```

RETURNS

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

DESCRIPTION

Sets scheduling options for the BTI429_SchedBuild function. The parameter *modeval* controls various options such as the scheduling method, time base and execution environment of the BTI429_SchedBuild function. The following is a list of the different values that can be OR'ed together for the *modeval* options. Only one METHOD may be specified per call.

modeval	
Constant	**Description**
SCHEDMODE_DEFAULT	Selects all default settings (**bold** below)
SCHEDMODE_METHOD_NORMAL	Selects the normal scheduling method
SCHEDMODE_METHOD_QUICK	Selects the quick scheduling method
SCHEDMODE_METHOD_BOTH	Uses the quick method first then uses the normal method if the quick method fails to build a schedule.
SCHEDMODE_MILLISEC	Sets the min and max periods to be specified in milliseconds
SCHEDMODE_MICROSEC	Sets the min and max periods to be specified in microseconds
SCHEDMODE_REMOTE	For RPC Devices, specifies to perform calculations remotely
SCHEDMODE_LOCAL	For RPC Devices, specifies to perform calculations locally

The normal scheduling method uses Ballard's traditional algorithm for scheduling messages. It is the default method to preserve schedule timing in established applications. The quick scheduling method significantly reduces the schedule build times and improves the ability to schedule high-duty cycle schedules. It is recommended to use the SCHEDMODE_METHOD_BOTH which first uses the quick scheduling method, and then uses the normal scheduling method if necessary.

For RPC Devices, SCHEDMODE_REMOTE and SCHEDMODE_LOCAL control where the scheduling calculations are executed. SCHEDMODE_REMOTE can be faster for many schedules or when network latency is high. SCHEDMODE_LOCAL can offer performance improvements for more complex schedules and if network latency is low.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

Calling this function sets the operational mode for BTI429_SchedBuild for ALL Devices in the same process space (i.e. same application). To use different schedule modes on different channels and/or different hardware, the BTI429_SchedMode function must be called prior to each subsequent call to BTI429_SchedBuild.

SEE ALSO

BTI429_SchedBuild

SchedMsg

```
SCHNDX BTI429_SchedMsg(
   MSGADDR message,          //Address of message
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a MESSAGE command Block to the current end of the Schedule. When a MESSAGE command Block is encountered in the Schedule, the message value specified by *message* is loaded into the transmit channel's encoder. The message will be transmitted when the Schedule subsequently encounters either a GAP Command Block or another MESSAGE command Block. If a MESSAGE command Block is followed by another MESSAGE command Block, the gap time between the two messages is automatically set to four bit times. More or less gap time can be explicitly set with BTI429_SchedGap or BTI429_SchedMsgEx.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function. In addition, the message must have been created with BTI429_MsgCreate.

SEE ALSO

BTI429_SchedMsgEx, BTI429_MsgCreate

SchedMsgEx

```
SCHNDX BTI429_SchedMsgEx(
   MSGADDR message,          //Address of message
   USHORT gap,               //Gap value to follow message
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a MESSAGE command Block to the current end of the Schedule with a user-specified gap value that can be less than the automatic minimum of 4 bit times. A MESSAGE command Block loads the message value specified by *message* into the transmit channel's encoder. The message will be transmitted when the Schedule encounters either a GAP Command Block or another MESSAGE command Block.

gap specifies the gap value (0 to 65,535) in bit times to follow the message. This gap value allows the user to explicitly specify a gap value of less (or more) than four bit times. This gap value is overwritten if followed by a GAP Command Block.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function. In addition, the message must have been created with BTI429_MsgCreate.

SEE ALSO

BTI429_SchedMsg, BTI429_MsgCreate

SchedPause

```
SCHNDX BTI429_SchedPause(
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Appends a PAUSE Command Block to the current end of the Schedule. A PAUSE Command Block pauses the channel specified by *channel* until the Device is unpaused using BTI429_ChResume. When a PAUSE Command Block is encountered, it has the same effect as executing the BTI429_ChPause function.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

Because of internal prefetching, when scheduling a SchedPause entry two very small preceding gaps (SchedGap with a gapval of 1) should be inserted. If these additional gaps are not inserted, the schedule could pause before the preceding message is transmitted.

SEE ALSO

BTI429_ChPause, BTI429_ChResume, BTI429_SchedHalt, BTI429_SchedRestart

SchedPulse

```
SCHNDX BTI429_SchedPulse(
   INT dionum,              //Dio number to pulse
   INT channel,             //Channel number
   HCORE hCore              //Core handle
)
```

RETURNS

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

DESCRIPTION

Schedules a pair of opcodes that pulse the specified *dionum* to the "On" state followed by the "Off" state. Use BTICard_ExtDIOWr for normal updating of the discrete I/O signals.

DEVICE DEPENDENCY

The level of the *dionum* "On" and "Off" states, as well as the mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

The SCHNDX returned is for the second of the pair of opcodes that are scheduled. The opcodes are scheduled sequentially, so the SCHNDX of the first pulse opcode can be derived from the return value by subtracting one.

Schedules internally use prefetching to minimize intermessage gap time. Because of schedule prefetching and asynchronous discrete operation, the absolute timing of the pulse from a SchedPulse entry can vary by a gap/message transmission. When comparing the occurrence of scheduled discrete transitions with adjacent 429 messages, the discrete can appear to transition early.

For discrete output pulses synchronous to ARINC 429 databus activity, use BTI429_ChSyncDefine or BTI429_MsgSyncDefine.

SEE ALSO

BTI429_SchedPulse0, BTI429_SchedPulse1, BTICard_ExtDIORd, BTICard_ExtDIOWr

SchedPulse0

```
SCHNDX BTI429_SchedPulse0(
   INT dionum,              //DIO number to pulse
   INT channel,             //Channel number
   HCORE hCore              //Core handle
)
```

RETURNS

The index of the appended schedule entry if successful, or a negative value if an error occurred.

DESCRIPTION

Sets the state of the specified *dionum* to the "Off" state. Use BTICard_ExtDIOWr for normal updating of the discrete I/O signals.

Note that by using this function in combination with BTI429_SchedGap, BTI429_SchedMessage and/or other schedule entries, the I/O signals can be used to frame messages, groups of messages, or to create arbitrary pulse width signals.

DEVICE DEPENDENCY

The level of the *dionum* "On" and "Off" states, as well as the mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

Schedules internally use prefetching to minimize intermessage gap time. Because of schedule prefetching and asynchronous discrete operation, the absolute timing of the pulse from a SchedPulse entry can vary by a gap/message transmission. When comparing the occurrence of scheduled discrete transitions with adjacent 429 messages, the discrete can appear to transition early.

For discrete output pulses synchronous to ARINC 429 databus activity, use BTI429_ChSyncDefine or BTI429_MsgSyncDefine.

SEE ALSO

BTI429_SchedPulse, BTI429_SchedPulse1, BTICard_ExtDIORd, BTICard_ExtDIOWr

SchedPulse1

```
SCHNDX BTI429_SchedPulse1(
   INT dionum,              //DIO number to pulse
   INT channel,             //Channel number
   HCORE hCore              //Core handle
)
```

RETURNS

The index of the appended schedule entry if successful, or a negative value if an error occurred.

DESCRIPTION

Sets the state of the specified *dionum* to the "On" state. Use BTICard_-ExtDIOWr for normal updating of the discrete I/O signals.

Note that by using this function in combination with BTI429_SchedGap, BTI429_SchedMessage and/or other schedule entries, the I/O signals can be used to frame messages, groups of messages, or to create arbitrary pulse width signals.

DEVICE DEPENDENCY

The level of the *dionum* "On" and "Off" states, as well as the mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device.

WARNINGS

Schedules internally use prefetching to minimize intermessage gap time. Because of schedule prefetching and asynchronous discrete operation, the absolute timing of the pulse from a SchedPulse entry can vary by a gap/message transmission. When comparing the occurrence of scheduled discrete transitions with adjacent 429 messages, the discrete can appear to transition early.

For discrete output pulses synchronous to ARINC 429 databus activity, use BTI429_ChSyncDefine or BTI429_MsgSyncDefine.

SEE ALSO

BTI429_SchedPulse, BTI429_SchedPulse0, BTICard_ExtDIORd, BTI-Card_ExtDIOWr

SchedRestart

```
SCHNDX BTI429_SchedRestart(
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTI429_SchedEntry

SchedReturn

```
SCHNDX BTI429_SchedReturn(
   INT channel,              //Channel number of transmitter
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

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

DEVICE DEPENDENCY

Does not apply to 5G Devices.

WARNINGS

A call to BTI429_ChConfig must precede this function.

SEE ALSO

BTI429_SchedCall

SeqBlkRd

```
ULONG BTICard_SeqBlkRd(
   LPUSHORT seqbuf,          //Pointer to Sequential Record buffer
   ULONG bufcount,           //Size of the buffer (in 16-bit words)
   LPULONG blockcount,       //Number of records copied to the buffer
   HCORE hCore               //Core handle
)
```

RETURNS

The number of 16-bit words copied to the user-supplied buffer.

DESCRIPTION

Copies as many available complete records as possible from the Sequential Record on the core to a buffer (*seqbuf*). The function returns the number of 16-bit words copied and the puts the number of records copied into *blockcount*. The larger the buffer size (*bufcount*) the greater the number of records that can be copied by a single call to this function. The data read is effectively removed from the Sequential Record on the core. This allows an infinite amount of data to be gathered as long as this function (or one of the others in the table below) is called frequently enough to prevent the Sequential Record on the core from overflowing.

There are four functions that read from the Sequential Record. BTICard_SeqRd reads a single record; BTICard_SeqBlkRd, BTICard_SeqCommRd, and BTICard_SeqDMARd read multiple records. Any one of these functions may be used in most applications. The difference lies in their speed of execution under different conditions and availability by Device. The table below compares the four functions that read from the Sequential Record and gives some rationale for selecting one over another:

Function	Function Overhead	Per Record Overhead	Use When…
BTICard_SeqRd	low	n/a	Expect one or no records per function call
BTICard_SeqBlkRd	low	high	Expect a small number of records per function call
BTICard_SeqCommRd	high	low	Expect a large number of records per function call
BTICard_SeqDMARd	low	low	Expect a large number of records per function call (Device-dependent)

DEVICE DEPENDENCY

On 3G Devices, BTICard_SeqRd, _SeqBlkRd, and _SeqCommRd all read multiple records in the same manner.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqCommRd, BTICard_SeqDMARd, BTICard_SeqBlkRd, BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqCommRd

```
USHORT BTICard_SeqCommRd(
   LPUSHORT seqbuf,          //Pointer to Sequential Record buffer
   USHORT bufcount,          //Size of the buffer (in 16-bit words)
   HCORE hCore               //Core handle
)
```

RETURNS

The number of 16-bit words copied to the user-supplied buffer.

DESCRIPTION

Copies as many available complete records as possible from the Sequential Record on the core to a buffer (*seqbuf*) and returns the number of 16-bit words copied. The larger the buffer size (*bufcount*) the greater the number of records that can be copied by a single call to this function. The data read is effectively removed from the Sequential Record on the core. This allows an infinite amount of data to be gathered as long as this function (or one of the others in the table below) is called frequently enough to prevent the Sequential Record on the core from overflowing.

There are four functions that read from the Sequential Record. BTICard_SeqRd reads a single record; BTICard_SeqBlkRd, BTICard_SeqCommRd, and BTICard_SeqDMARd read multiple records. Any one of these functions may be used in most applications. The difference lies in their speed of execution under different conditions and availability by Device. The table below compares the four functions that read from the Sequential Record and gives some rationale for selecting one over another:

Function	Function Overhead	Per Record Overhead	Use When…
BTICard_SeqRd	low	n/a	Expect one or no records per function call
BTICard_SeqBlkRd	low	high	Expect a small number of records per function call
BTICard_SeqCommRd	high	low	Expect a large number of records per function call
BTICard_SeqDMARd	low	low	Need to offload application from reading monitor data (Device-dependent)

DEVICE DEPENDENCY

On 3G Devices, BTICard_SeqRd, _SeqBlkRd, and _SeqCommRd all read multiple records in the same manner.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqBlkRd, BTICard_SeqDMARd, BTICard_SeqRd, BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqConfig

```
ERRVAL BTICard_SeqConfig(
   ULONG ctrlflags,          //Selects configuration options
   HCORE hCore               //Core handle
)
```

RETURNS

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

DESCRIPTION

Configures the Sequential Monitor of the core by allocating an on-board buffer and initializing internal pointers associated with the buffer.

ctrlflags	
Constant	**Description**
SEQCFG_DEFAULT	Select all default settings (**bold** below)
SEQCFG_DISABLE	Disable Sequential Record
SEQCFG_DMA	Enables DMA mode (Device-dependent)
SEQCFG_FILLHALT	Enable Sequential Record in fill and halt mode
SEQCFG_CONTINUOUS	Enable Sequential Record in continuous mode
SEQCFG_DELTA	Enable Sequential Record in delta mode (ARINC 429 only)
SEQCFG_INTERVAL	Enable Sequential Record in interval mode (ARINC 429 only)
SEQCFG_16K	Allocate a 16 K Sequential Record buffer
SEQCFG_32K	Allocate a 32 K Sequential Record buffer
SEQCFG_64K	Allocate a 64 K Sequential Record buffer
SEQCFG_128K	Allocate a 128 K Sequential Record buffer
SEQCFG_ALLAVAIL	Allocate all available memory to the Sequential Record
SEQCFG_NOLOGFULL	Do not generate an entry in the Event Log List when the Sequential Record is full
SEQCFG_LOGFULL	Generate an entry in the Event Log List when the Sequential Record is full
SEQCFG_NOLOGFREQ	Do not generate entries in the Event Log List at user-defined frequency (see BTICard_SeqLogFrequency)
SEQCFG_LOGFREQ	Generate entries in the Event Log List at user-defined frequency (see BTICard_SeqLogFrequency)

Note: It is highly recommended that the SEQCFG_FILLHALT mode be used for the Sequential Record. This mode will allow for continuous recording of databus activity as long as the host keeps up with reading out record data. To allow the host flexibility in reading the Sequential Record, it is also recommended to use a value of SEQCFG_128K for the size of the buffer.

When using the SEQCFG_CONTINOUS mode, databus activity will be continuously written to the Sequential Record without regard for the host reading data from the buffer. If the host attempts to read from it while the Device is running, the data returned could be corrupted. Therefore, when in this mode the Sequential Record should only be read while stopping and resuming the monitor using BTICard_SeqStop and BTICard SeqResume.

DEVICE DEPENDENCY

5G Devices always have a 16MB Sequential Record buffer. 5G 429 Devices do not support interval and delta mode. Sequential DMA is supported on Windows and Linux Hosts for 4G and 5G Devices, and embedded Linux for 5G RPC Devices.

WARNINGS

If the SEQCFG_ALLAVAIL flag is used, BTICard_SeqConfig should be the last function called that allocates memory before BTICard_CardStart is called.

SEE ALSO

BTICard_SeqRd, BTICard_SeqInterval, BTICard_SeqLogFrequency

SeqDMARd

```
USHORT BTICard_SeqDMARd(
   LPUSHORT seqbuf,          //Pointer to Sequential Record buffer
   ULONG bufcount,           //Size of the buffer (in 16-bit words)
   HCORE hCore               //Core handle
)
```

RETURNS

The number of 16-bit words copied to the user-supplied buffer.

DESCRIPTION

Copies as many available complete records as possible from the Sequential Record on the core to the buffer *seqbuf* and returns the number of 16-bit words copied. The larger the buffer size (*bufcount*) the greater the number of records that can be copied by a single call to this function. The Sequential Record data is read from Host memory since it was already transferred from the core memory to the Host in a DMA process. This allows an infinite amount of data to be gathered as long as this function (or one of the others in the table below) is called frequently enough to prevent the Sequential Record on the core from overflowing.

There are four functions that read from the Sequential Record. BTICard_SeqRd reads a single record; BTICard_SeqBlkRd, BTICard_SeqCommRd, and BTICard_SeqDMARd read multiple records. Any one of these functions may be used in most applications. The difference lies in their speed of execution under different conditions and availability by Device. The table below compares the four functions that read from the Sequential Record and gives some rationale for selecting one over another:

Function	Function Overhead	Per Record Overhead	Use When…
BTICard_SeqRd	low	n/a	Expect one or no records per function call
BTICard_SeqBlkRd	low	High	Expect a small number of records per function call
BTICard_SeqCommRd	high	Low	Expect a large number of records per function call
BTICard_SeqDMARd	low	Low	Need to offload application from reading monitor data (Device-dependent)

DEVICE DEPENDENCY

Applies only to 4G and 5G Devices. Sequential DMA is supported on Windows and Linux Hosts, and embedded Linux for 5G RPC Devices.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqBlkRd, BTICard_SeqRd, BTICard_SeqDMARd, BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqFindCheckVersion

```
BOOL BTICard_SeqFindCheckVersion(
   LPUSHORT pRecord,          //Pointer to a record
   USHORT version,            //Version number to test
)
```

RETURNS

TRUE if record pointed to by *pRecord* is equal to or greater than the version number represented by *version*, otherwise FALSE.

DESCRIPTION

Checks to see if the version number of the record pointed to by *pRecord* is equal to or greater than the constant passed for *version*. Use this function to test the eligibility of a given record for a version-dependent application of a BTICard_SeqFindMore?? function.

version	
Constant	**Description**
SEQVER_0	Sequential Record Version 0
SEQVER_1	Sequential Record Version 1

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqFindInit

```
ERRVAL BTICard_SeqFindInit(
   LPUSHORT seqbuf,          //Pointer to a Sequential Record buffer
   ULONG seqcount,           //Number of 16-bit words in the buffer
   LPSEQFINDINFO sfinfo      //Pointer to structure
)
```

RETURNS

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

DESCRIPTION

Initializes the structure (*sfinfo*) used by other BTICard_SeqFindNext?? functions for finding records within a Sequential Record buffer. *seqbuf* is a pointer to the start of a buffer containing Sequential Record data, and *seqcount* is the number of 16-bit words in the buffer.

sfinfo contains information that is used by the various BTICard_SeqFindNext?? functions. Each time a BTICard_SeqFindNext?? function is called, the *sfinfo* structure is updated to indicate where to resume the search with the next BTICard_SeqFindNext?? function. Since these find functions pick up where they left off, based on *sfinfo*, it is necessary to call BTICard_SeqFindInit whenever a find function is to start at the beginning of the buffer.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

This function causes the BTICard_SeqFindNext?? functions to start their search at the beginning of the Sequential Record buffer.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindNext??

SeqFindMore1553

```
ERRVAL BTICard_SeqFindMore1553(
   LPSEQRECORDMORE1553 *pRecMore,   //Address of pointer to a structure
   LPSEQRECORD1553 pRecBase         //Pointer to a structure
)
```

RETURNS

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

DESCRIPTION

Finds the extra fields at the end of a MIL-STD-1553 record in the Sequential Record buffer pointed to by *pRecBase* and updates **pRecMore* to point to those fields. BTICard_SeqFindNext1553 must be called before each call to BTICard_SeqFindMore1553 to seed the *pRecBase* structure with the first portion of a 1553 message. Repeatedly calling BTICard_SeqFindNext1553 and BTICard_SeqFindMore1553 returns the 1553 records in the Sequential Record one at a time until the end of the buffer is reached (at which time BTICard_SeqFindNext1553 returns an error value).

Part of the time-tag and the measured RT response times are recorded in the extra fields, as shown in the table below.

SEQRECORDMORE1553 structure			
Field	**Size**	**Description**	**Version (of base record)**
timestamph	ULONG	Upper 32 bits of the time-tag value	1 or greater
resptime1	USHORT	First RT response time (in 10ths of µs)	1 or greater
resptime2	USHORT	Second RT response time (in 10ths of µs)	1 or greater

DEVICE DEPENDENCY

Applies only to 4G and 5G Devices, which add extra fields to 1553 records of version 1 or greater. The version number of the base record pointed to by *pRecBase* can be tested using BTICard_SeqFindCheckVersion.

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

BTICard_SeqFindNext1553 must be called before each call to BTICard_SeqFindMore1553.

SEE ALSO

BTICard_SeqFindInit, BTICard_SeqFindNext1553

SeqFindNext

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

RETURNS

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

DESCRIPTION

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

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

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

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

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqFindNext1553

```
ERRVAL BTICard_SeqFindNext1553(
   LPSEQRECORD1553 *pRecord,     //Address of pointer to a structure
   LPSEQFINDINFO sfinfo          //Pointer to structure
)
```

RETURNS

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

DESCRIPTION

Finds the next MIL-STD-1553 record in the Sequential Record buffer and updates **pRecord* to point to that record. This function uses and updates data in the *sfinfo* structure. Calling this function repeatedly returns the 1553 records one at a time until the end of the buffer is reached (at which time it returns an error value). Thus, messages can be individually saved to disk, displayed on screen, etc. Using the predefined Sequential Record structure SEQRECORD1553 allows for easy handling of the data.

SEQRECORD1553 structure			
Field	**Size**	**Description**	**Version**
type	USHORT	The protocol and version number of this record	All versions
timestamp	ULONG	Lower 32 bits of the time-tag value	All versions
activity	USHORT	Activity value (see table below for details)	All versions
error	USHORT	Error value (see table below for details)	All versions
cwd1	USHORT	Command word 1 value	All versions
cwd2	USHORT	Command word 2 value	All versions
swd1	USHORT	Status word 1value	All versions
swd2	USHORT	Status word 2 value	All versions
datacount	USHORT	Number of MIL-STD-1553 data words	All versions
data[]	USHORT	Array of 1553 data words (don't exceed data[datacount -1])	All versions

The version number can be tested using BTICard_SeqFindCheckVersion.

The activity and error fields may be tested by AND-ing the values returned with the constants from the tables below:

MIL-STD-1553 activity field	
Constant	**Description**
MSGACT1553_CHMASK	The channel number. Shift the result right by MSGACT-1553_CHSHIFT.
MSGACT1553_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACT1553_XMTCWD1	Command word 1 was transmitted.
MSGACT1553_XMTCWD2	Command word 2 was transmitted.
MSGACT1553_XMTSWD1	Status word 1 was transmitted.
MSGACT1553_XMTSWD2	Status word 2 was transmitted.
MSGACT1553_RCVCWD1	Command word 1 was received.
MSGACT1553_RCVCWD2	Command word 2 was received.
MSGACT1553_RCVSWD1	Status word 1 was received.
MSGACT1553_RCVSWD2	Status word 2 was received.
MSGACT1553_XMTDWD	Data word was transmitted.
MSGACT1553_RCVDWD	Data word was received.
MSGACT1553_BUS	Message was transmitted/received on bus A (0) or B (1).

MIL-STD-1553 error field	
Constant	**Description**
MSGERR1553_NORESP	No response was received from the RT
MSGERR1553_ANYERR	Set if any other error bits are set
MSGERR1553_PROTOCOL	A protocol error occurred
MSGERR1553_SYNC	Wrong polarity of the sync pulse
MSGERR1553_DATACOUNT	Too many/too few data words
MSGERR1553_MANCH	Manchester error
MSGERR1553_PARITY	Parity error
MSGERR1553_WORD	Word error
MSGERR1553_RETRY	All attempts to retry transmission of this message failed (BC only)
MSGERR1553_SYSTEM	Internal system error occurred
MSGERR1553_HIT	Indicates that this message was transmitted or received since this bit was last cleared (always set)

Extract the channel number from the activity word by AND-ing the activity field with MSGACT1553_CHMASK and right-shifting the result by MSGACT1553_CHSHIFT. The resulting value is the channel number associated with the MIL-STD-1553 record.

```
channel = (activity & MSGACT1553_CHMASK) >> MSGACT1553_CHSHIFT;
```

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Must be preceded by a call to BTICard_SeqFindInit.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit, SeqFindMore-1553

SeqFindNext429

```
ERRVAL BTICard_SeqFindNext429(
   LPSEQRECORD429 *pRecord,   //Address of pointer to a structure
   LPSEQFINDINFO sfinfo       //Pointer to structure
)
```

RETURNS

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

DESCRIPTION

Finds the next ARINC 429 record in the Sequential Record buffer and updates **pRecord* to point to that record. This function uses and updates data from the *sfinfo* structure. Calling this function repeatedly returns the 429 records one at a time until the end of the buffer is reached (at which time it returns an error value). Thus, messages can be individually saved to disk, displayed on screen, etc. Using the predefined Sequential Record structure SEQRECORD429 allows for easy handling of the data.

SEQRECORD429 structure

Field	Size	Description	Version
type	USHORT	The protocol and version number of this record	All versions
timestamp	ULONG	Lower 32 bits of the time-tag value	All versions
activity	USHORT	Activity (see table below for details)	All versions
decgap	USHORT	Gap preceding the 429 word in half bit-times (up to a maximum of 7.5 bit times)	1 or greater
data	ULONG	32-bit ARINC 429 data word value	All versions
timestamph	ULONG	Upper 32 bits of the time-tag value	1 or greater

The version number can be tested using BTICard_SeqFindCheckVersion.

The *decgap* field is a 4-bit value that measures the gap preceding the decoded word. If *decgap* indicates 7.5 bit times (F hexadecimal), then the gap is 7.5 bit times or greater.

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

ARINC 429 activity field

Constant	Description
MSGACT429_CHMASK	The channel number. Shift the result right by MSGACT429_CHSHIFT.
MSGACT429_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACT429_SPD	This bit reflects the speed detected. A one signifies high speed (100 Kbps), and a zero signifies low speed (12.5 Kbps).
MSGACT429_ERR	If set, it signifies that an error occurred in receiving this word. The type of error is defined by the following bits.
MSGACT429_GAP	Gap Error. A gap of less than four bit times preceded the word.
MSGACT429_PAR	Parity error. A parity error was detected in the word.
MSGACT429_LONG	Long word error. A word of more than 32-bits was detected.
MSGACT429_BIT	Bit timing error. An error occurred while decoding the bits of the word (short bits or long bits).
MSGACT429_TO	Time out error. The decoder timed out while receiving a word (short word).
MSGACT429_HIT	Signifies that the message has been processed by the firmware (the Hit bit).

Extract the channel number from the activity word by AND-ing the activity field with MSGACT429_CHMASK and right-shifting the result by MSGACT-429_CHSHIFT. The resulting value is the channel number associated with the ARINC 429 record.

```
channel = (activity & MSGACT429_CHMASK) >> MSGACT429_CHSHIFT;
```

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Must be preceded by a call to BTICard_SeqFindInit.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit

SeqFindNext708

```
ERRVAL BTICard_SeqFindNext708(
   LPSEQRECORD708 *pRecord, //Address of pointer to a structure
   LPSEQFINDINFO sfinfo     //Pointer to structure
)
```

RETURNS

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

DESCRIPTION

Finds the next ARINC 708 record in the Sequential Record buffer and updates **pRecord* to point to that record. This function uses and updates data from the *sfinfo* structure. Calling this function repeatedly returns the 708 records one at a time until the end of the buffer is reached (at which time it returns an error value). Thus, messages can be individually saved to disk, displayed on screen, etc. Using the predefined Sequential Record structure SEQRECORD708 allows for easy handling of the data.

SEQRECORD708 structure

Field	Size	Description	Version
type	USHORT	The protocol and version number of this record	All versions
timestamp	ULONG	Lower 32 bits of the time-tag value	All versions
activity	USHORT	Activity (see table below for details)	All versions
datacount	USHORT	Number of data words	All versions
data[100]	USHORT	ARINC 708 data word values (100 16-bit data words)	All versions
extra[16]	USHORT	Additional data (if variable bit mode is enabled)	1 or greater
bitcount	USHORT	Number of bits in this message (if variable bit mode is enabled)	1 or greater
timestamph	ULONG	Upper 32 bits of the time-tag value	1 or greater

The version number can be tested using BTICard_SeqFindCheckVersion.

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

ARINC 708 activity field

Constant	Description
MSGACT708_CHMASK	The channel number. Shift the result right by MSGACT708_CHSHIFT.
MSGACT708_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACT708_ERR	This bit is set if any of the error bits are set.
MSGACT708_DSYNC	No data sync at end of word.
MSGACT708_MANCH	Manchester error.
MSGACT708_WORD	Word error.
MSGACT708_LONG	Long word error. A word of more than 1600 bits was detected.
MSGACT708_SHORT	Short word error. A word of less than 1600 bits was detected.
MSGACT708_TO	Time out error. The decoder timed out while receiving a word.
MSGACT708_HIT	Indicates that this message was transmitted or received since this bit was last cleared.

Extract the channel number from the activity word by AND-ing the activity field with MSGACT708_CHMASK and right-shifting the result by MSGACT-708_CHSHIFT. The resulting value is the channel number associated with the ARINC 708 record.

```
channel = (activity & MSGACT708_CHMASK) >> MSGACT708_CHSHIFT;
```

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Must be preceded by a call to BTICard_SeqFindInit.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit

SeqFindNext717

```
ERRVAL BTICard_SeqFindNext717(
   LPSEQRECORD717 *pRecord, //Address of pointer to a structure
   LPSEQFINDINFO sfinfo     //Pointer to structure
)
```

RETURNS

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

DESCRIPTION

Finds the next ARINC 717 record in the Sequential Record buffer and updates **pRecord* to point to that record. This function uses and updates data from the *sfinfo* structure. Calling this function repeatedly returns the 717 records one at a time until the end of the buffer is reached (at which time it returns an error value). Thus, messages can be individually saved to disk, displayed on screen, etc. Using the predefined Sequential Record structure SEQRECORD717 allows for easy handling of the data.

SEQRECORD717 structure

Field	Size	Description	Version
type	USHORT	The protocol and version number of this record	All versions
timestamp	ULONG	Lower 32-bits of the time-tag value	All versions
activity	USHORT	Activity (see table below for details)	All versions
wordnum	USHORT	Number of words	All versions
subframe	USHORT	Number of subframes	All versions
superframe	USHORT	Number of superframes	All versions
data	USHORT	12-bit ARINC 717 data word value in LSBs	All versions
timestamph	ULONG	Upper 32 bits of the time-tag value	1 or greater

The version number can be tested using BTICard_SeqFindCheckVersion.

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

ARINC 717 activity field

Constant	Description
MSGACT717_CHMASK	The channel number. Shift the result right by MSGACT717_CHSHIFT.
MSGACT717_CHSHIFT	Channel number shift value. See CHMASK above.
MSGACT717_SPDMASK	The current speed mask value.
MSGACT717_64WPS	The current speed is 64 wps (words per second).
MSGACT717_128WPS	The current speed is 128 wps.
MSGACT717_256WPS	The current speed is 256 wps.
MSGACT717_512WPS	The current speed is 512 wps.
MSGACT717_1024WPS	The current speed is 1024 wps.
MSGACT717_2048WPS	The current speed is 2048 wps.
MSGACT717_4096WPS	The current speed is 4096 wps.
MSGACT717_8192WPS	The current speed is 8192 wps.
MSGACT717_TO	Time out error. The decoder timed out while receiving a (short) word.
MSGACT717_HIT	Indicates that this message was transmitted or received since this bit was last cleared.

Extract the channel number from the activity word by AND-ing the activity field with MSGACT717_CHMASK and right-shifting the result by MSGACT-717_CHSHIFT. The resulting value is the channel number associated with the ARINC 717 record.

```
channel = (activity & MSGACT717_CHMASK) >> MSGACT717_CHSHIFT;
```

DEVICE DEPENDENCY

When IRIG is enabled on a 4G Device, time-tags in message structures and Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Must be preceded by a call to BTICard_SeqFindInit.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit

SeqFindNextDIO

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

RETURNS

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

DESCRIPTION

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

SEQRECORDDIO structure			
Field	**Size**	**Description**	**Version**
type	USHORT	The protocol and version number of this record	All versions
count	USHORT	The length of this record	All versions
bank	USHORT	Number of the bank	All versions
state	USHORT	State of the bank	0 only
timestamp	ULONG	Lower 32 bits of the time-tag value	All versions
timestamph	ULONG	Upper 32 bits of the time-tag value	All versions
change	USHORT	Bitmask of discrete inputs that changed value	1 or greater
value	USHORT	Current value of discrete inputs	1 or greater

DEVICE DEPENDENCY

Applies to all 5G Devices. The mapping of *dionum* to physical discrete I/O is hardware dependent. Please consult the hardware manual for the Device. Also applies to 4G Devices with one or more Discrete I/O modules (832 module). Please consult the OmniBus Discrete IO User's Manual for usage with 4G Devices.

When IRIG is enabled on a 4G Device, time-tags in Sequential Records will be in BCD format (see BTICard_TimerStatus).

WARNINGS

Must be preceded by a call to BTICard_SeqFindInit.

SEE ALSO

BTICard_ExtDIOMonConfig, BTICard_SeqConfig, BTICard_SeqRd, BTICard_SeqFindInit

SeqInterval

```
INT BTICard_SeqInterval(
   INT interval,              //Interval time (in seconds)
   INT mode,                  //Mode to determine interval value
   HCORE hCore                //Core handle
)
```

RETURNS

The actual interval value that the core will use.

DESCRIPTION

Sets the interval time for the Sequential Monitor, and is used when the Sequential Record has been configured with the SEQCFG_INTERVAL flag. In Interval mode, the Sequential Monitor records only the first occurrence of selected messages within the specified interval. The availability of the Interval mode is both Device- and protocol-dependent.

The core cannot accommodate all interval values that could be passed through *interval*. The specified mode helps determine the actual interval that will be used. The constants below should be used to set the mode:

mode	
Constant	**Description**
INTERVALMODE_CLOSEST	Uses the value closest to the specified interval
INTERVALMODE_LESS	Uses the value just less than specified interval
INTERVALMODE_GREATER	Uses the value just greater the specified interval

DEVICE DEPENDENCY

Applies only to 3G and 4G Devices.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd

SeqIsRunning

```
BOOL BTICard_SeqIsRunning(
   HCORE hCore                    //Core handle
)
```

RETURNS

TRUE if the Sequential Record is still active, otherwise FALSE.

DESCRIPTION

Determines whether the Sequential Record is active and is typically used when the Sequential Record has been configured with the SEQCFG_FILLHALT flag. In which case, recording halts when the on-board Sequential Record is full. This prevents unread data from being overwritten when the host gets behind in reading data from the Sequential Record. Thus, in fill and halt mode BTICard_SeqIsRunning effectively indicates whether the buffer is full or not.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd

SeqLogFrequency

```
USHORT BTICard_SeqLogFrequency(
   USHORT logfreq,          //Frequency of Event Log List entries
   HCORE hCore              //Core handle
)
```

RETURNS

The previous value of the frequency of Event Log List entries.

DESCRIPTION

Sets the Event Log List frequency for the Sequential Monitor. It is used when the Sequential Record has been configured with the SEQCFG_LOGFREQ flag in BTICard_SeqConfig. The Sequential Record generates an Event Log List entry after it records *logfreq* amount of records. The user specifies the value of *logfreq*. For example, a value of 1 enables an Event Log List entry after every record, a value of 2 after every second record, and so on. It continues in this manner until the Sequential Record is stopped.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqRd, BTICard_EventLogConfig, BTICard_EventLogRd

SeqRd

```
USHORT BTICard_SeqRd(
   LPUSHORT seqbuf,          //Pointer to Sequential Record buffer
   HCORE hCore               //Core handle
)
```

RETURNS

The number of 16-bit words copied to the user-supplied buffer.

DESCRIPTION

Copies up to a single record at a time from the Sequential Record on the core to a buffer (*seqbuf*). The function returns the number of 16-bit words copied. The data read is effectively removed from the Sequential Record on the core. This allows an infinite amount of data to be gathered as long as this function (or one of the others in the table below) is called frequently enough to prevent the Sequential Record on the core from overflowing.

There are four functions that read from the Sequential Record. BTICard_SeqRd reads a single record; BTICard_SeqBlkRd, BTICard_SeqCommRd, and BTICard_SeqDMARd read multiple records. Any one of these functions may be used in most applications. The difference lies in their speed of execution under different conditions and availability by Device. The table below compares the four functions that read from the Sequential Record and gives some rationale for selecting one over another:

Function	Function Overhead	Per Record Overhead	Use When…
BTICard_SeqRd	low	n/a	Expect one or no records per function call
BTICard_SeqBlkRd	low	High	Expect a small number of records per function call
BTICard_SeqCommRd	high	Low	Expect a large number of records per function call
BTICard_SeqDMARd	low	Low	Need to offload application from reading monitor data (Device-dependent)

DEVICE DEPENDENCY

On 3G Devices, BTICard_SeqRd, BTICard_SeqBlkRd, and BTICard_SeqCommRd all read multiple records in the same manner.

WARNINGS

None.

SEE ALSO

BTICard_SeqConfig, BTICard_SeqBlkRd, BTICard_SeqCommRd, BTICard_SeqFindInit, BTICard_SeqFindNext??

SeqResume

```
BOOL BTICard_SeqResume(
   HCORE hCore                    //Core handle
)
```

RETURNS

TRUE if the Sequential Record was previously running, otherwise FALSE.

DESCRIPTION

Resumes operation of the Sequential Record at the point at which it was stopped using BTICard_SeqStop. Use this function to continue recording data to the Sequential Record without overwriting previous records.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

A call to BTICard_SeqStop must precede this function.

SEE ALSO

BTICard_SeqStart, BTICard_SeqStop

SeqStart

```
BOOL BTICard_SeqStart(
   HCORE hCore                  //Core handle
)
```

RETURNS

TRUE if the Sequential Record was previously running, otherwise FALSE.

DESCRIPTION

Starts recording of the Sequential Record. If necessary, it also stops and clears the Sequential Record before restarting it.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

If this function is called after BTICard_SeqStop, recording starts at the beginning of the buffer and previous data is overwritten. To add to previous data without erasing it, use BTICard_SeqResume instead.

SEE ALSO

BTICard_SeqStop, BTICard_SeqResume

SeqStatus

```
BOOL BTICard_SeqStatus(
   HCORE hCore                    //Core handle
)
```

RETURNS

The status value of the Sequential Record.

DESCRIPTION

Checks the status of the Sequential Record. The status value can be tested using the predefined constants below:

Constant	Description
STAT_EMPTY	Sequential Record is empty
STAT_PARTIAL	Sequential Record is partially filled
STAT_FULL	Sequential Record is full
STAT_OFF	Sequential Record is off

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

The operation of the SeqStatus is configuration and Device dependent. All 4G Devices and 5G RPC Devices configured for DMA can return STAT_EMPTY when the internal sequential buffer is empty but the DMA buffer contains data.

SEE ALSO

BTICard_SeqStart, BTICard_SeqStop, BTICard_SeqResume

SeqStop

```
BOOL BTICard_SeqStop(
   HCORE hCore                    //Core handle
)
```

RETURNS

TRUE if the Sequential Record was previously running, otherwise FALSE.

DESCRIPTION

Suspends the recording of data to the Sequential Record before the buffer is filled. If BTICard_SeqResume is subsequently called, recording is resumed at the point at which it was stopped without overwriting previous records. If BTICard_SeqStart is called after BTICard_SeqStop, recording starts at the beginning of the buffer and previous data is overwritten.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_SeqStart, BTICard_SeqResume

SysMonClear

```
ERRVAL BTICard_SysMonClear(
   HCARD hCard                    //Card handle
)
```

RETURNS

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

DESCRIPTION

Resets the historic maximum and minimum values for all sensors on the card specified by *hCard*.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_SysMonTypeGet

SysMonDescGet

```
LPCSTR BTICard_SysMonDescGet(
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

A pointer to a character string describing the sensor or NULL if the sensor is not present.

DESCRIPTION

Provides a formatted string that describes the sensor specified by *index*.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_SysMonTypeGet

SysMonMaxRd

```
INT BTICard_SysMonMaxRd(
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

The historic maximum value of a sensor or SYSMONRD_NOTVALID if the sensor is not present or the historic maximum value is not valid.

DESCRIPTION

Reads the historic maximum value from the sensor specified by *index*. The value is in units of mV, mA, or m°C depending on the sensor type. Call BTICard_SysMonUserStr to convert the value to a formatted string.

To reset the historic maximum value, call BTICard_SysMonClear.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

This function will return SYSMONRD_NOTVALID for up to 800 ms after the first call to BTICard_CardOpen and up to 150 ms after any call to BTICard_SysMonClear to allow SysMon to get valid data.

Note: Due to occasionally spurious values that may be reported by the **voltage/current** sensor, a second read of the system monitor to confirm an error is recommended prior to software acting on the assumed error data. See BTICard_SysMonValRd for additional information.

SEE ALSO

BTICard_SysMonClear, BTICard_SysMonMinRd, BTICard_SysMonUserStr

SysMonMinRd

```
INT BTICard_SysMonMinRd(
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

The historic minimum value of a sensor or SYSMONRD_NOTVALID if the sensor is not present or the historic minimum value is not valid.

DESCRIPTION

Reads the historic minimum value from the sensor specified by *index*. The value is in units of mV, mA, or m°C depending on the sensor type. Call BTICard_SysMonUserStr to convert the value to a formatted string.

To reset the historic minimum value, call BTICard_SysMonClear.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

This function will return SYSMONRD_NOTVALID for up to 800ms after the first call to BTICard_CardOpen and up to 150ms after any call to BTICard_SysMonClear to allow SysMon to get valid data.

Note: Due to occasionally spurious values that may be reported by the **voltage/current** sensor, a second read of the system monitor to confirm an error is recommended prior to software acting on the assumed error data. See BTICard_SysMonValRd for additional information.

SEE ALSO

BTICard_SysMonClear, BTICard_SysMonMaxRd, BTICard_SysMonUserStr, BTICard_SysMonValRd

SysMonNomRd

```
INT BTICard_SysMonNomRd(
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

The nominal voltage for a voltage sensor or SYSMONRD_NOTVALID if the sensor is not present.

DESCRIPTION

Reads the nominal voltage from the voltage sensor specified by *index*. The value is in units of mV. Call BTICard_SysMonUserStr to convert the value to a formatted string.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

Does not apply to temperature and current sensors.

SEE ALSO

BTICard_SysMonUserStr, BTICard_SysMonValRd

SysMonThresholdGet

```
ERRVAL BTICard_SysMonThresholdGet(
   BOOL *enable,    //Pointer to variable to receive enable value
   LPINT minval,    //Pointer to variable to receive minimum threshold value
   LPINT maxval,    //Pointer to variable to receive maximum threshold value
   INT index,       //Specifies the sensor index
   HCARD hCard      //Card handle
)
```

RETURNS

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

DESCRIPTION

Reads the user definable thresholds, in units of m°C, for the temperature sensor specified by *index*. These user thresholds are disabled and set to System Limits at power-on and can be modified by calling BTICard_SysMonThresholdSet. The enable value and thresholds are passed through *enable*, *minval*, and *maxval* respectively.

Call BTICard_SysMonUserStr to convert *minval* or *maxval* to a formatted string.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

Due to rounding, *minval* and *maxval* may not match what was set in BTICard_-SysMonThresholdSet. The values may be rounded to the nearest resolution.

Does not apply to voltage and current sensors.

SEE ALSO

BTICard_BITStatusRd, BTICard_SysMonThresholdSet

SysMonThresholdSet

```
ERRVAL BTICard_SysMonThresholdSet(
   BOOL enable,              //Enable for the Sensor
   INT minval,               //Minimum threshold value
   INT maxval,               //Maximum threshold value
   INT index,                //Specifies the sensor index
   HCARD hCard               //Card handle
)
```

RETURNS

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

DESCRIPTION

Sets the user definable thresholds, in units of m°C, for the temperature sensor specified by *index*. The enable value and thresholds are passed through *enable*, *minval*, and *maxval* respectively. Once enabled, if the sensor value exceeds the the user definable thresholds, the BIT Status register will indicate a fault which can be read by calling BTICard_BITStatusRd.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

Due to rounding, *minval* and *maxval* may not match what was set in BTICard_-SysMonThresholdSet. The values may be rounded to the nearest resolution.

Does not apply to voltage and current sensors.

SEE ALSO

BTICard_BITStatusRd, BTICard_SysMonThresholdSet

SysMonTypeGet

```
ULONG BTICard_SysMonTypeGet(
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

The type of sensor or SYSMONTYPE_NONE if the sensor is not present.

DESCRIPTION

Reports the type of the sensor specified by *index* by returning one of the predefined constants below:

Constant	Description
SYSMONTYPE_NONE	Sensor is not present
SYSMONTYPE_TEMP	Temperature Sensor
SYSMONTYPE_VOLTAGE	Voltage Sensor
SYSMONTYPE_CURRENT	Current Sensor

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_SysMonDescGet

SysMonUserStr

```
LPCSTR BTICard_SysMonUserStr(
   INT value,                  //Value to be formatted
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

A pointer to a character string containing the value and units for a sensor or NULL if sensor is not present.

DESCRIPTION

Returns a formatted character string containing the value and units for the sensor specified by *index*. The parameter *value* is typically read by calling BTICard_SysMonValRd, BTICard_SysMonNomRd, BTICard_SysMonMinRd, or BTICard_SysMonMaxRd.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

None.

SEE ALSO

BTICard_SysMonValRd, BTICard_SysMonNomRd, BTICard_SysMonMinRd, BTICard_SysMonMaxRd

SysMonValRd

```
INT BTICard_SysMonValRd(
   INT index,                  //Specifies the sensor index
   HCARD hCard                 //Card handle
)
```

RETURNS

The current value for a sensor or SYSMONRD_NOTVALID if the sensor is not present.

DESCRIPTION

Reads the current value of the sensor specified by *index*. The units for the current value are in mV, mA, or m°C depending on the sensor type. Call BTICard_SysMonUserStr to convert the value to a formatted string.

DEVICE DEPENDENCY

Applies to 5G Devices that support BIT/SysMon functionality. Please consult the hardware manual for the Device.

WARNINGS

This function will return SYSMONRD_NOTVALID for up to 800ms after the first call to BTICard_CardOpen and up to 150ms after any call to BTICard_SysMonClear to allow SysMon to get valid data.

Note: Due to occasionally spurious values that may be reported by the **voltage/current** sensor, a second read of the system monitor to confirm an error is recommended prior to software acting on the assumed error data. A minimum wait time of 150 ms is required before issuing a second read of the system monitor to ensure that a new value has been sampled. Call BTICard_SysMonClear to clear min/max values if a spurious voltage or current values is read.

SEE ALSO

BTICard_SysMonUserStr, BTICard_SysMonMaxRd, BTICard_SysMonMinRd

Timer64Rd

```
ERRVAL BTICard_Timer64Rd(
   LPULONG valueh,          //Pointer to upper 32 bits of the timer value
   LPULONG valuel,          //Pointer to lower 32 bits of the timer value
   HCORE hCore              //Core handle
)
```

RETURNS

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

DESCRIPTION

Reads the current value of the binary timer from the specified Device.

DEVICE DEPENDENCY

Applies to all Devices. 3G and 4G Devices have a 32-bit binary timer, while 5G Devices have a 48-bit binary timer. To use the IRIG timer for a specified 4G Device instead of the default binary timer, see BTICard_IRIGConfig. Please see BTICard_TimerRd for a discussion of Device-dependent timer differences.

WARNINGS

None.

SEE ALSO

BTICard_TimerClear, BTICard_TimerRd, BTICard_TimerWr, BTICard_-Timer64Wr, BTICard_IRIGConfig, BTICard_IRIGRd, BTICard_IRIGWr

Timer64Wr

```
VOID BTICard_Timer64Wr(
   ULONG valueh,              //Upper 32 bits of the timer value
   ULONG valuel,              //Lower 32 bits of the timer value
   HCORE hCore                //Core handle
)
```

RETURNS

None.

DESCRIPTION

Writes the timer value of the binary timer for the specified Device.

DEVICE DEPENDENCY

Applies to all Devices. 3G and 4G Devices have a 32-bit binary timer, while 5G Devices have a 48-bit binary timer. To use the IRIG timer for a specified 4G Device instead of the default binary timer, see BTICard_IRIGConfig. Please see BTICard_TimerWr for a discussion of Device-dependent timer differences.

WARNINGS

None.

SEE ALSO

BTICard_TimerClear, BTICard_TimerRd, BTICard_TimerWr, BTICard_-Timer64Rd, BTICard_IRIGConfig, BTICard_IRIGRd, BTICard_IRIGWr

TimerClear

```
VOID BTICard_TimerClear(
   HCORE hCore                    //Core handle
)
```

RETURNS

None.

DESCRIPTION

Clears the Device timer to zero.

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_TimerRd, BTICard_TimerResolution, BTICard_IRIGConfig, BTICard_IRIGRd, BTICard_IRIGWr

TimerRd

```
ULONG BTICard_TimerRd(
   HCORE hCore                    //Core handle
)
```

RETURNS

The current Device timer value.

DESCRIPTION

Reads the lower 32-bits of the current value of the binary timer from the specified Device.

DEVICE DEPENDENCY

Applies to all Devices.

5G Devices have a 48-bit binary timer with a 1 µs resolution.

For 3G and 4G Devices, the binary timer exists in two parts: a hardware DSP timer, and a software extended value. Together these values make a 48-bit time value. BTICard_TimerResolution is used to adjust which bits of this 48-bit value are used to make the 32-bit time-tag used in message structures and Sequential Records. BTICard_TimerRd and BTICard_TimerWr functions only read from and write to the software extended portion of this time value and not the hardware portion (due to the complexity of accounting for the rollover from the Host, an accurate reading of both the hardware and software part is not possible). The software extended portion used by BTICard_TimerRd and BTICard_TimerWr has a resolution of 4.096 ms.

To use the IRIG timer on a 4G Device to generate time-tag values for message structures and Sequential Records instead of the default binary timer, see BTICard_IRIGConfig.

WARNINGS

None.

SEE ALSO

BTICard_Timer64Rd, BTICard_Timer64Wr, BTICard_TimerClear, BTICard_TimerResolution, BTICard_TimerWr, BTICard_IRIGConfig, BTICard_IRIGRd, BTICard_IRIGWr

TimerResolution

```
INT BTICard_TimerResolution(
   INT timerresol,          //Selects the timer resolution
   HCORE hCore              //Core handle
)
```

RETURNS

The value of the previous resolution.

DESCRIPTION

Selects the resolution for the time-tag timer on the specified Device. *timerresol* must be one of the following predefined constants:

timerresol		
Constant	**Resolution**	**Range (hr:min:sec)**
TIMERRESOL_1US	1 µs	1:11:34
TIMERRESOL_16US	16 µs	19:05:19
TIMERRESOL_1024US	1024 µs	50 days

DEVICE DEPENDENCY

Applies only to 3G and 4G Devices. The 5G binary timer resolution is always 1 µs, and has a range of 365 days. To use the IRIG timer for a specified 4G core instead of the default binary timer, see BTICard_IRIGConfig.

WARNINGS

After changing the resolution, a call to BTICard_TimerClear should be made to clear the timer.

SEE ALSO

BTICard_TimerClear, BTICard_IRIGConfig, BTICard_IRIGRd, BTICard_-IRIGWr

TimerStatus

```
INT BTICard_TimerStatus(
   HCORE hCore                  //Core handle
)
```

RETURNS

The status value of the timer configuration.

DESCRIPTION

This function determines the status of how the timer for a core is configured. Some Devices have configurable modes that affect elements of data structures. For example, 4G Devices allow for BCD or binary time-tag formatting.

The status value can be tested using the predefined constants below:

Constant	Description
TIMETAG_FORMAT_BCD	Time-tags are in BCD format
TIMETAG_FORMAT_BIN	Time-tags are in binary format

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

3G Devices and 4G Devices (when configured to use binary formatted time-tags) latch the time-tag value when processed by the Device firmware. This will create some minor variability in time-tags from message to message. 5G Devices and 4G Devices (when configured to use IRIG BCD formatted time-tags) latch the time-tag value when processed by the Device hardware. This results in a very consistent and accurate time-tag value.

Some ARINC 429 messages of 4G Devices (when configured for binary formatted time-tags) may have identical time-tags due to batch processing of messages in the same Device firmware time slot. This is most noticeable in the Sequential Record when comparing time-tags of messages.

SEE ALSO

BTICard_IRIGConfig, BTICard_IRIGTimeBCDToBin, BTICard_IRIGTimeBinToBCD, BTICard_SeqFind??

TimerWr

```
VOID BTICard_TimerWr(
   ULONG value,                  //Value of the timer
   HCORE hCore                   //Core handle
)
```

RETURNS

None.

DESCRIPTION

Writes *value* to the lower 32-bits of the binary timer of the specified Device.

DEVICE DEPENDENCY

Applies to all Devices.

5G Devices have a 48-bit binary timer with a 1 µs resolution.

For 3G and 4G Devices, the binary timer exists in two parts: a hardware DSP timer, and a software extended value. Together these values make a 48-bit time value. BTICard_TimerResolution is used to adjust which bits of this 48-bit value are used to make the 32-bit time-tag used in message structures and Sequential Records. BTICard_TimerRd and BTICard_TimerWr functions only read from and write to the software extended portion of this time value and not the hardware portion (due to the complexity of accounting for the rollover from the Host, an accurate reading of both the hardware and software part is not possible). The software extended portion used by BTICard_TimerRd and BTICard_TimerWr has a resolution of 4.096 ms.

To use the IRIG timer on a 4G Device to generate time-tag values for message structures and Sequential Records instead of the default binary timer, see BTICard_IRIGConfig.

WARNINGS

None.

SEE ALSO

BTICard_Timer64Wr, BTICard_Timer64Rd, BTICard_TimerRd, BTICard_TimerResolution, BTICard_IRIGConfig, BTICard_IRIGRd, BTICard_IRIGWr

ValFromAscii

```
ULONG BTICard_ValFromAscii(
   LPCSTR asciistr,          //ASCII string to convert
   INT radixval              //Radix of string
)
```

RETURNS

The converted integer numeric value.

DESCRIPTION

Converts a string representation of a 32-bit value with the specified radix to an integer. Processing stops at the first null terminator. *radixval* can be any positive integer, but is commonly 16 for hexadecimal, 8 for octal, or 10 for decimal.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

No check is made for invalid characters.

SEE ALSO

BTICard_ValToAscii

ValGetBits

```
ULONG BTICard_ValGetBits(
   ULONG oldvalue,          //The old value
   INT startbit,            //Position of starting bit of field
   INT endbit               //Position of ending bit of field
)
```

RETURNS

The value of the extracted bit field.

DESCRIPTION

Extracts the specified bit field from the 32-bit integer *oldvalue*. The result is obtained by masking the field and shifting the *endbit* to the LSB of the return value. The LSB is bit number 0.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_ValPutBits

ValPutBits

```
USHORT BTICard_ValPutBits(
   ULONG oldvalue,          //The old value
   ULONG newfld,            //The value of the new field
   INT startbit,            //Position of starting bit of field
   INT endbit               //Position of ending bit of field
)
```

RETURNS

The integer value with the inserted bit field.

DESCRIPTION

Inserts a bit field into a 32-bit integer value. The *oldval* is masked and OR-ed with the shifted value of *newfld*. The LSB is bit number 0.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_ValGetBits

ValToAscii

```
LPSTR BTICard_ValToAscii(
   ULONG value,              //The value to be converted
   LPSTR asciistr,           //A string to receive the results
   INT numbits,              //The number of significant bits
   INT radixval,             //The radix value
)
```

RETURNS

An ASCII string representing the integer.

DESCRIPTION

Creates a string representation of an integer in a specified radix. The string is copied to *asciistr* and is also returned. The string is always null-terminated. *asciistr* is assumed to be large enough to hold the resulting string.

The length of the string is determined by *numbits* and *radixval* and is padded by leading zeros. *radixval* can be any positive integer but is commonly 16 for hexadecimal, 8 for octal, or 10 for decimal. For example, a string representation of a value with 16 significant bits and a radix of 16 will always be 4 characters long followed by a null terminator.

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

DEVICE DEPENDENCY

Applies to all Devices.

WARNINGS

None.

SEE ALSO

BTICard_ValFromAscii

This page intentionally blank.

APPENDIX B: MULTI-PROTOCOL / DEVICE PROGRAMS

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

Programming Rules

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

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

BTICard_ Functions

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

Sequential Record

Each core has one Sequential Record, independent of how many different protocols it supports. The format of individual records within the Sequential Record differs between protocols. There are two ways of scanning through a Sequential Record: by protocol-specific records or by every record. To scan by protocol, use the BTICard_SeqFindNext?? function to find the next record with the ?? protocol. For instance, the BTICard_SeqFindNext429 and BTICard_SeqFindNext1553 functions are used to find the next ARINC 429 or MIL-STD-1553 record respectively. To scan through every record, use the BTICard_SeqFindNext function, which finds the next record and returns the type (429, 1553, etc.) of the record it found. The different BTICard_SeqFindNext?? functions should not be mixed within a sequence without first using BTICard_SeqFindInit. Note that the BTICard_SeqFindNext?? functions do not use a handle, so they do not access the Device. They work from a copy of the Sequential Record in the computer's memory. Thus, they may be used to process a Sequential Record that had been previously saved to a hard disk.

Event Log List

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

Using Multiple Devices

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

APPENDIX C: REVISION HISTORY

The following revisions have been made to this manual:

Rev A Date: July 12th, 2001

Original release of this manual.

Rev B Date: January 23rd, 2009

Major revision of both the text and the appendices to accommodate the features and functionality of multi-core and 5G Devices. Many functions are new or modified.

Rev. B.1 Date: August 7, 2009

Minor revision of both the text and appendices to fix typos and add clarifications to function usage.

Rev. C Date: January 6, 2010

Major revision of both the text and the appendices to accommodate the features and functionality of USB 429 Devices.

Rev. D Date: June 14, 2011

Major revision of both the text and the appendices to accommodate the features and functionality of Mx5 Devices. Added BIT and SysMon functions.

Rev E. Date: November 21, 2013

Minor revision of both the text and the appendices to accommodate SysMon function warnings and list function warning. Added MsgMultiSkipWr and SchedPulse functions. Added AB3000, PM1553-5, USB 708, and USB Multi Device references. Some function descriptions modified.

This page intentionally blank.