# Beagle™ Protocol Analyzers

The Beagle™ Protocol Analyzers are non-intrusive bus monitors which allow developers to see and analyze serial bus data in real time as it appears on the bus. All Beagle analyzers feature:

- Monitor packets in real-time as they appear on the bus
- High-speed USB up-link to analysis computer
- Windows, Linux, and Mac OS X compatible

### Beagle USB 5000 SuperSpeed Protocol Analyzer v2 Features

- Non-intrusive super-, high-, full-, and low-speed USB monitoring
- 2 GB on-board hardware buffer, up to 4 GB supported
- Digital inputs and outputs for synchronizing with external logic
- Packet-level timing down to 2 ns resolution
- Advanced Match/Action triggers and filters
- Data scrambling, spread-spectrum clocking, and receiver detection
- Cross-analyzer synchronization

### Beagle USB 480 Protocol Analyzer Series Features

- Non-intrusive high-, full-, and low-speed USB monitoring
- Large 64 MB on-board hardware buffer (256 MB in Power models)
- Digital inputs and outputs for synchronizing with external logic
- Packet-level timing with 16.6 ns resolution
- $V_{BUS}$ Current/Voltage Monitoring (Power models only)
- Advanced Match/Action triggers and filters (Power model Ultimate Edition only)

### Beagle USB 12 Protocol Analyzer Features

- Non-intrusize full-, low-speed USB monitoring (12 and 1.5 Mbps)
- Bit-level timing with 21 ns resolution

### Beagle {{i2c]]}}/SPI/MDIO Protocol Analyzer Features

- Non-intrusize {i2c]] monitoring up to 4 MHz
- Non-intrusize SPI monitoring up to 24 MHz
- Non-intrusize MDIO monitoring up to 2.5 MHz
- User selectable bit-level timing (up to 20 ns resolution)

TOTAL PHASE

Supported products:

BEAGLE™ PROTOCOL ANALYZER

Beagle Protocol Analyzers
User Manual v5.10
November 27, 2013

# 1 General Overview

## 1.1 USB Background

### 1.1.1 USB History

Universal Serial Bus (USB) is a standard interface for connecting peripheral devices to a host computer. The USB system was originally devised by a group of companies including Compaq, Digital Equipment, IBM, Intel, Microsoft, and Northern Telecom to replace the existing mixed connector system with a simpler architecture.

USB was designed to replace the multitude of cables and connectors required to connect peripheral devices to a host computer. The main goal of USB was to make the addition of peripheral devices quick and easy. All USB devices share some key characteristics to make this possible. All USB devices are self-identifying on the bus. All devices are hot-pluggable to allow for true Plug'n'Play capability. Additionally, some devices can draw power from the USB which eliminates the need for extra power adapters.

To ensure maximum interoperability the USB standard defines all aspects of the USB system from the physical layer (mechanical and electrical) all the way up to the software layer. The USB standard is maintained and enforced by the USB Implementers Forum (USB-IF). USB devices must pass a USB-IF compliance test in order to be considered in compliance and to be able to use the USB logo.

USB 1.0 was first introduced in 1996, but was not adopted widely until 1998 with USB 1.1. In 2000, USB 2.0 was released and has since become the de facto standard for connecting devices to computers and beyond. In 2008, the USB specification was expanded with USB 3.0, also known as SuperSpeed USB. USB 3.0 represents a significant change in the underlying operation of USB. To simplify the experience for the user, USB 3.0 has been designed to be plug-n-play backwards compatible with USB 2.0.

USB 3.0 specification include a number of significant changes including:

- Higher data transfer rate (up to 5 Gbps)
- Increased bus power and current draw
- Improved power management
- Full duplex data communications
- Link Training and Status State Machine (LTSSM)
- Interrupt driven, instead of polling

- Streaming interface for more efficient data transfers

As of 2010, the USB standard specifies different flavors of USB: low-speed, full-speed, high-speed, and SuperSpeed. USB-IF has also released additional specs that expand the breadth of USB. These are On-The-Go (OTG) and Wireless USB. Although beyond the scope of this document, details on these specs can be found on the USB-IF website.

### 1.1.2 Architectural Overview

USB is a host-scheduled, token-based serial bus protocol. USB allows for the connection of up to 127 devices on a single USB host controller. A host PC can have multiple host controllers which increases the maximum number of USB devices that can be connected to a single computer.

Devices can be connected and disconnected at will. The host PC is responsible for installing and uninstalling drivers for the USB devices on an as-needed basis.

A single USB system comprises of a USB host and one or more USB devices. There can also be zero or more USB hubs in the system. A USB hub is special class of device. The hub allows the connection of multiple downstream devices to an upstream host or hub. In this way, the number of devices that can be physically connected to a computer can be increased.

A USB device is a peripheral device that connects to the host PC. The range of functionality of USB devices is ever increasing. The device can support either one function or many functions. For example a single multi-function printer may present several devices to the host when it is connected via USB. It can present a printer device, a scanner device, a fax device, etc.

All the devices on a single USB must share the bandwidth that is available on the bus. It is possible for a host PC to have multiple buses which would all have their own separate bandwidth. Most often, the ports on most motherboards are paired, such that each bus has two downstream ports.

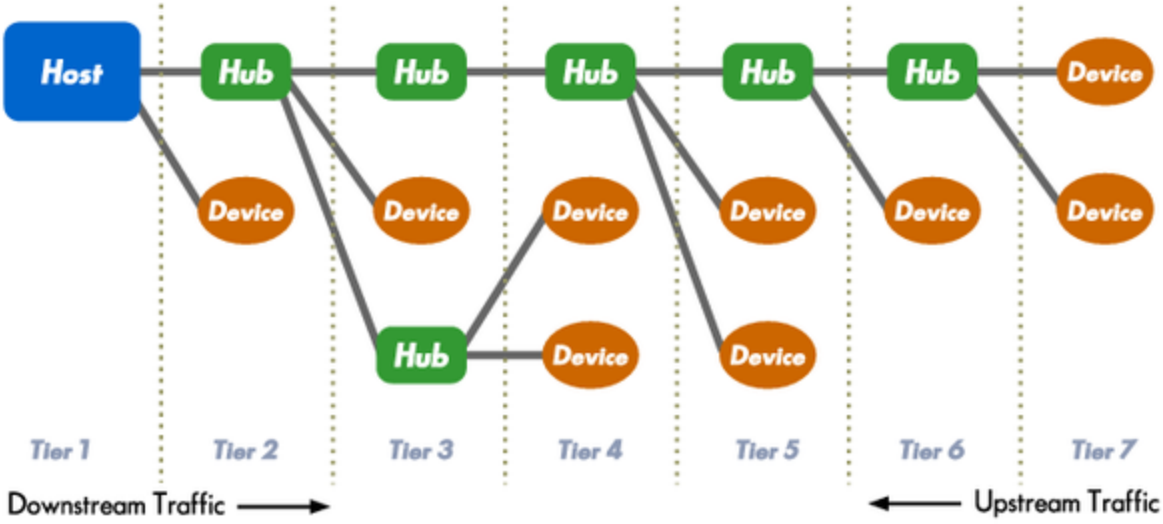


***Figure 1*** *: Sample USB Bus Topology.*
*A USB can only have a single USB host device. This host can support up to 127 different devices on a single port. There is an upper limit of 7 tiers of devices which means that a maximum of 5 hubs can be connected inline.*

The USB has a tiered star topology (Figure 1 ). At the root tier is the USB host. All devices connect to the host either directly or via a hub. According to the USB spec, a USB host can only support a maximum of seven tiers.

### USB 2.0 Specific Architecture

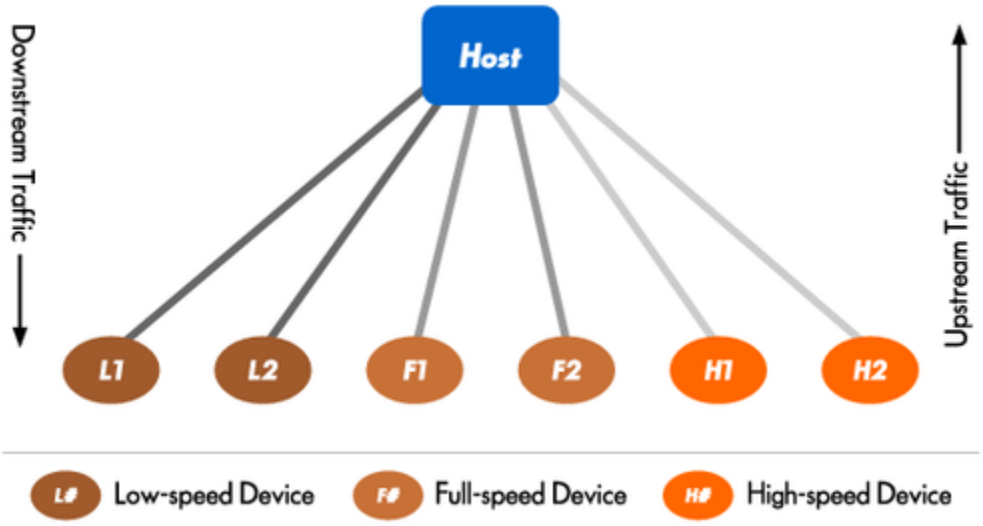

***Figure 2** : USB Broadcast*
*A USB 2.0 host broadcasts information to all the devices below it. Low-speed and high-speed enabled devices will only see traffic at their respective speeds. Full-speed devices can see both their speed and low-speed traffic.*

USB 2.0 works through a unidirectional broadcast system. When a host sends a packet, all downstream devices will see that traffic. If the host wishes to communicate with a specific device, it must include the address of the device in the token packet. Upstream traffic (the response from devices) are only seen by the host or hubs that are directly on the return path to the host.

There are, however, a few caveats when dealing with devices that are of different speeds. Low-speed and high-speed devices are isolated from traffic at speeds other then their own. They will only see traffic that is at their respective speeds. Referring to Figure 2, this means that downstream traffic to device H1 will be seen by device H2 (and vice versa). Also, downstream traffic to device L1 will be seen by L2 (and vice versa). However, full-speed devices can see traffic at its own speed, as well as low-speed traffic, using a special signaling mode dubbed low-speed-over-full-speed. This means that downstream traffic to F1 will be seen by F2 (and vice versa) with standard full-speed signaling, and downstream traffic to either L1 or L2 will also be seen by both F1 and F2 through the special low-speed-over-full-speed signaling.

## USB 3.0 Specific Architecture

USB 3.0 marks a significant change from the existing USB infrastructure and affects the protocol at nearly all levels. The major features of USB 3.0 will be covered briefly in this datasheet. For detailed information please consult the USB specifications from the USB-IF.

### USB 3.0 Physical Interface

Due to limitations of the differential signaling of USB 2.0, in order to be able to support 5 Gbps data communications, the physical interface has been upgraded. In addition to the normal USB 2.0 signals, USB 3.0 cables and connectors have two additional pairs of differential signals: one pair for transmit and one pair for receive, as seen in Figure 3.

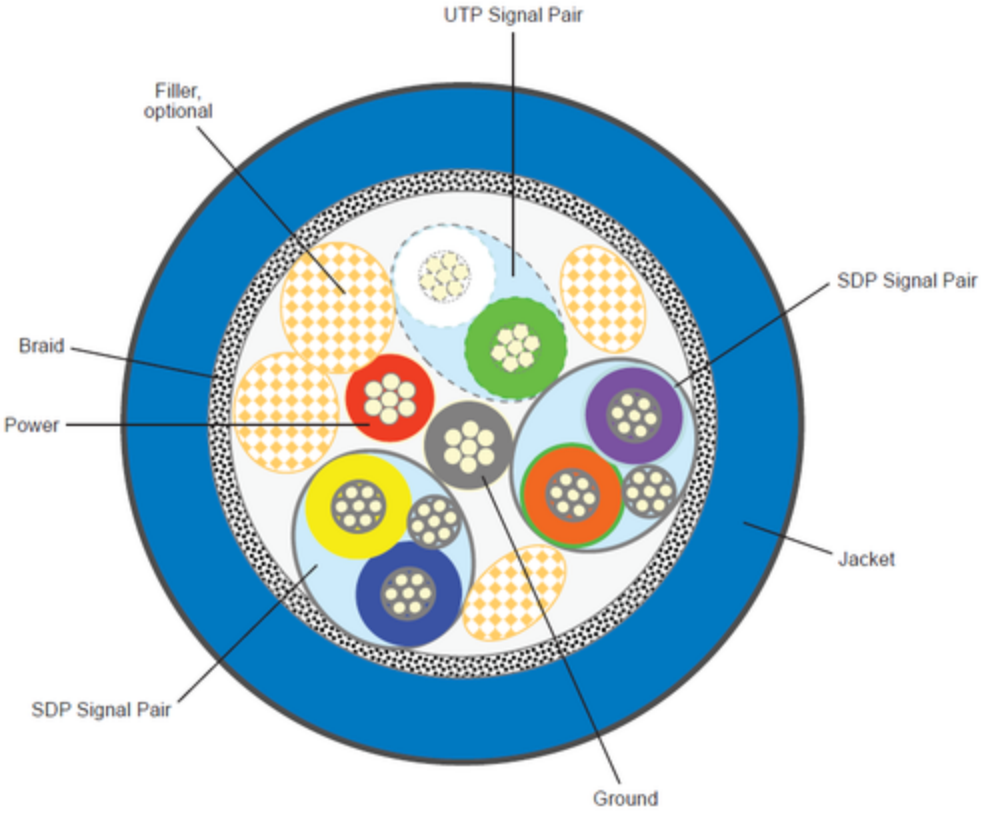


***Figure 3** : USB 3.0 Cable*
*Cross-section of a USB 3.0 cable. Image courtesy of USB Implementers Forum*

These two additional pairs allow for full-duplex communication over USB 3.0. Since the original USB 2.0 lines are unchanged, USB 2.0 communications can occur in parallel to USB 3.0.

**USB 3.0 Power**

Many of the key changes in USB 3.0 involve power and power management of USB devices.

**USB 3.0 Power Distribution**

The amount of power available to USB devices has been increased in USB 3.0. For unconfigured devices, 150 mA of power is available, compared with only 100 mA of power in USB 2.0. 150 mA is considered one unit load. Configured devices are able to draw up to 6 unit loads, or 900 mA, a significant increase from the 500 mA available in USB 2.0. The added power allows for a broader range of devices to be bus-powered.

**USB 3.0 Power Management**

USB 3.0 provides better power management facilities to use power more efficiently, and to help reduce overall power consumption.

Link-level power management allows the host or device to initiate a transition to a lower-level power state. There are three low power states available that are shown in Figure 4.

In USB 3.0, there is no longer periodic device polling and packets are no longer broadcast on the bus. It is now possible for devices to enter low-power states when idle in USB 3.0 because they no longer have to manage the reception of these packets.

Low-power levels are configurable on the device level and the function level. A device can suspend all or some of this functionality when it is idle, therefore reducing its power consumption.

With Latency Tolerance Messaging, devices can report their latency tolerance to the host, allowing the host system to enter lower power states without negatively affecting the USB devices on the bus.

**USB 3.0 Physical Layer**

In USB 3.0, the physical layer specifies the electrical characteristics of SuperSpeed USB signals how information is scrambled and encoded, and special signal sequences used by other layers.

Here is a brief overview of some of the new technologies specific to SuperSpeed USB.

**Receiver Termination**

USB 3.0 receivers terminate the transmission line by placing a small resistor to ground. Transmitters will check for this termination resistor on the receiver as a way for detecting the presence of a USB 3.0 receiver.

**Data Scrambling**

The physical layer uses bit scrambling to reduce electrical interference problems on the lines. However, it is possible for a transmitter to disable this feature.

**8b/10b encoding**

8b/10b encoding maps 8-bit symbols to 10-bit symbols with the purpose of keeping a low disparity while continuing to have enough edge transitions for clock recovery.

Disparity is kept low by taking advantage of the increased number space that 10b has to offer. Since all the 8b values would only take a subset of the 10b number space, multiple 10b symbols can be used encode a single 8b value. Often times, two different 10b symbols will be used to encode an 8b value, where the two 10b symbols have different number of 1s and 0s. The 10b symbol that is chosen to be sent will minimize the existing disparity on the line, with the goal of having a net 50/50 distribution of 1s and 0s. For

example, if a line has a running disparity of +2 1s, the next symbol on the line will have a bit pattern that has more 0s.

In addition, the increased number space allows for the use of certain control symbols, called K symbols which do not map to any 8b data value. USB 3.0 uses these control symbols for a number of purposes including: packet framing, elastic buffer mitigation, and data scrambler control.

**Training Sequences**

To accomodate for the various signaling characteristics of all manufactured transmitters, cables, and connectors, SuperSpeed receivers must be trained upon connection to a transmitter. This training sequence helps configure the receiver equalization, polairty, and data scrambler in order to establish a successful communications link.

**Spread Spectrum Clocking**

SuperSpeed USB employs spread spectrum clocking on its signaling. The advantage of this is that rather than radiating all energy in a small frequency band at a high level, a spread spectrum clock spreads its energy in a slightly larger frequency band, which reduces the peak level at any specific frequency. This is done to help meet EMC regulations.

**Low-Frequency Periodic Signaling (LFPS)**

LFPS signal is a side-band of communication sent on the normal SuperSpeed data lines at a lower frequency (10-50 MHz instead of 5 Gbps). This side-band helps to manage signal initiation and low power management on the bus on a link between two ports.

**Elastic Buffer**

USB 3.0 devices do not share the exact same clock source. Therefore they must be able to tolerate small variations between reference clocks on the transmitter and receiver. To compensate for such differences, receivers implement elasticity buffers that add or throw away dummy data, called SKP ordered sets, based on the state of the buffer at the time that the SKP ordered sets were received.

**USB 3.0 Link Layer**

The Link Layer is responsible for establishing and maintaining a reliable channel between a host and a device. There are a number of key concepts in this layer:

**Link Commands**

Link Commands are used to ensure the successful transfer of a packet, link flow control, and link power management.

**Link Training and Status State Machine (LTSSM)**

LTSSM is a state machine that defines link connectivity and link power management. LTSSM consists of 12 states: 4 operational link states (U0, U1, U2, U3), 4 link initialization and training states (Rx.Detect, Polling, Recovery, Hot Reset), 2 link test states (Loopback and Compliance Mode), SS Inactive (which is a link error state where USB 3.0 is non-operable) and SS.Disabled (where the SuperSpeed bus is disabled and operates as USB 2.0 only). Figure 4 maps out all the states of LTSSM and defines how the link transitions between states.

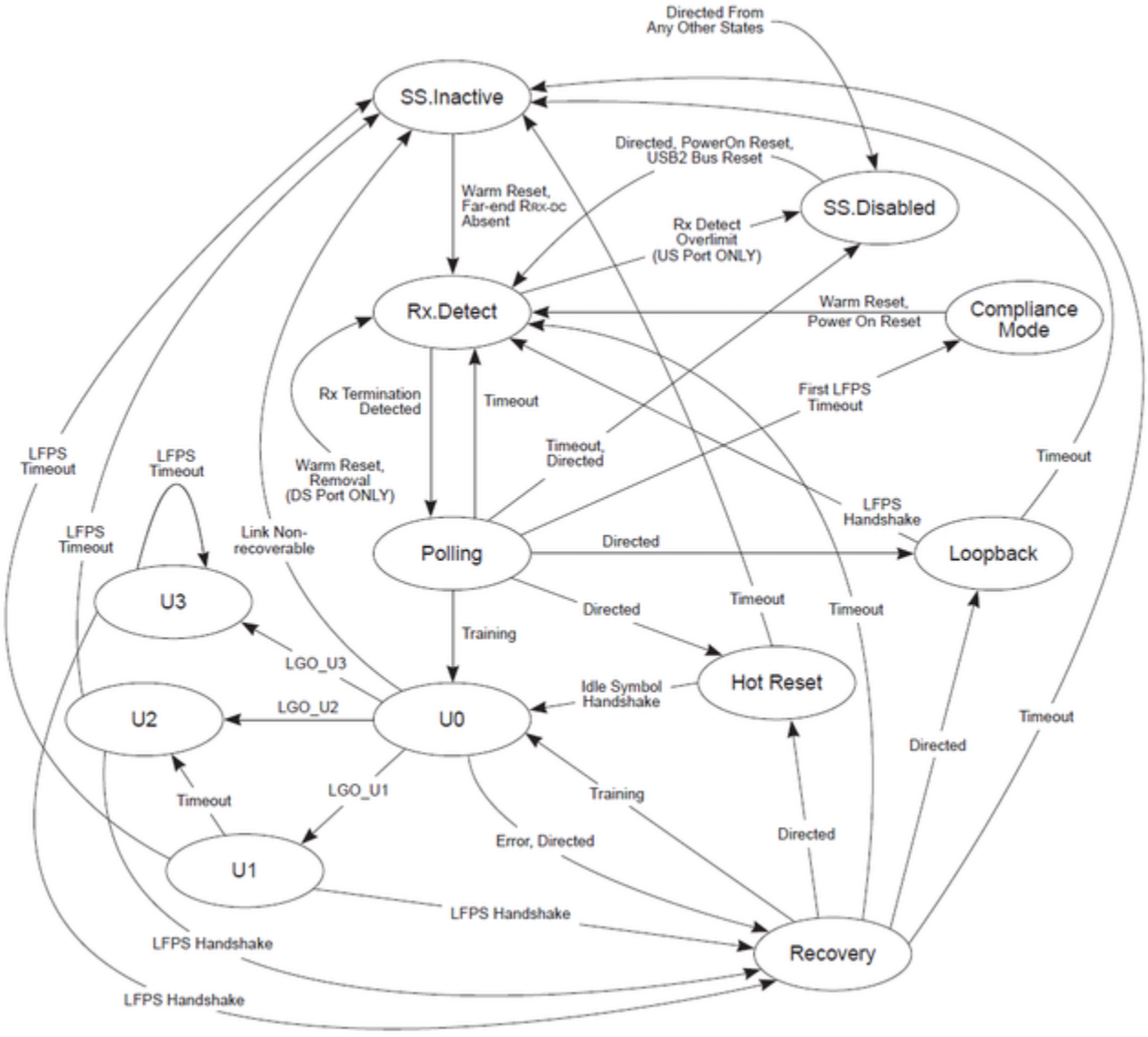

***Figure 4** : LTSSM State Machine*
*The Link Training and Status State Machine (LTSSM) is the core of the USB 3.0 link layer and defines link connectivity and link power management states states and transitions.*
*Image courtesy of USB Implementers Forum*

In order for a USB 3.0 device to enter the U0 operational link state, the link must be trained in order to synchronize the transmitter and receiver between the host and device.

Key LTSSM link states:

**Rx.Detect**

This is the initial power-on state where a transmitter checks for proper receiver termination to determine if its SuperSpeed partner is present on the bus. When the termination is detected, link training can begin.

**Polling**

During the polling state, two link partners train the link to synchronize their communications in preparation for data transmission.

**U0**

This is the normal operational state where SuperSpeed signaling is enabled and 5Gb packets are transmitted and received.

**U1, U2, U3**

These are low-power states where no 5Gb packets are transmitted. U1, U2, and U3 have increasingly longer wakeup times into U0, and thus allow transmitters to go into increasingly deeper sleeps.

**USB 3.0 Protocol Layer**

The USB 3.0 protocol layer manages the flow of data between devices, and specifies how the different packet structures are used. USB 3.0 specific packets are shown in Figure 17.

### 1.1.3 Theory of Operations

This introduction is a general summary of the USB spec. Total Phase strongly recommends that developers consult the USB specification on the USB-IF website for detailed and up-to-date information.

### USB 2.0 Connectors

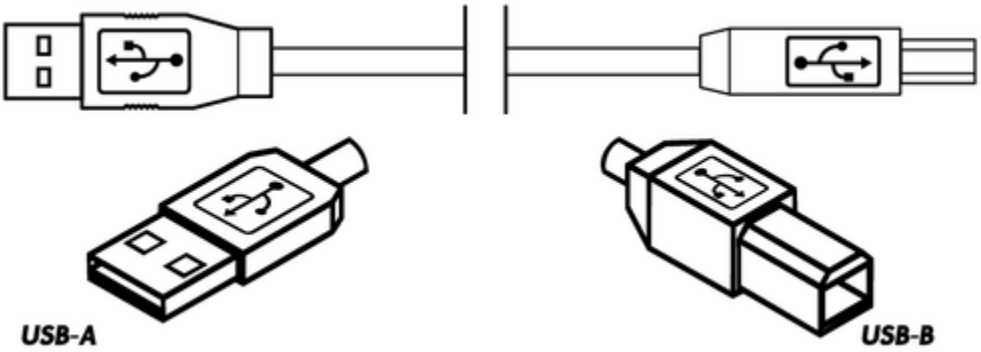


***Figure 5** : USB Cable*
*A USB cable has two different types of connectors: "A" and "B". The "A" connectors connect upstream towards the Host and B connectors connect downstream to the Devices.*

USB cables have two different types of connectors: "A" and "B". "A" type connectors connect towards the host or upstream direction. "B" connectors connect to downstream devices, though many devices have captive cables eliminating the need for "B" connectors. The "A" and "B" connectors are defined in the USB spec to prevent loopbacks in the bus. This prevents a host from being connected to a host, or conversely a device to a device. It also helps enforce the tiered star topology of the bus. USB hubs have one "B" port and multiple "A" ports which makes it clear which port connects to the host and which to downstream devices.

The USB spec has been expanded to include Mini-A and Mini-B connectors to support small USB devices. The USB On-The-Go (OTG) spec has introduced the Micro-A plug, Micro-B plug and receptacle, and the Micro-AB receptacle to allow for device-to-device connections. (The previous Mini-A plug and Mini-AB receptacle have now been deprecated.)

### USB 3.0 Connectors

The new USB 3.0 connectors serve two purposes. First, the connectors must be capable of physically interfacing with USB 3.0 signals to provide the ability to send and receive SuperSpeed USB data. Secondly, the connectors must be backwards compatible with USB 2.0 cables.

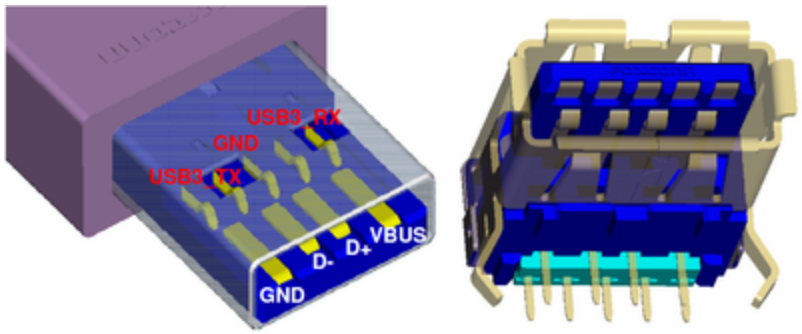


***Figure 6** : USB 3.0 Standard-A Connector*
*USB 3.0 Standard-A plug and receptacle. Image courtesy of USB Implementers Forum*

The USB 3.0 Standard-A connector (Figure 6 ) is very similar in appearance to the USB& 2.0 Standard-A connector. However, the USB 3.0 Standard-A connector and receptacle have 5 additional pins: a differential pair for transmitting data, a differential pair for receiving data, and the drain. USB 3.0 Standard-A plugs and receptacles are often colored blue to help differentiate it from USB 2.0.

The USB 3.0 Standard-A connector has been designed to be able to be plugged into either a USB t;2.0 or USB 3.0 receptacle. Similarly, the USB 3.0 Standard-A receptacle is designed to accept both the USB 3.0 and the USB ;2.0 Standard-A plugs.

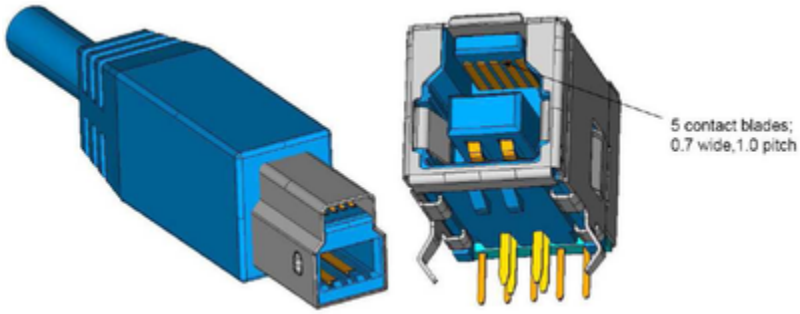


***Figure 7** : USB 3.0 Standard-B Connector*
*USB 3.0 Standard-B plug and receptacle. Image courtesy of USB Implementers Forum*

The USB 3.0 Standard-B connector (Figure 7 ) is similar to the USB ;2.0 Standard-B connector, with an additional structure at the top of the plug for the additional USB 3.0 pins. Due to the distinct appearance of the USB 3.0 Standard-B plug and receptacle, they do not need to be color coded, however many manufacturers color them blue to match the Standard-A connectors.

Given the new geometry, the USB 3.0 Standard-B plug is only compatible with USB 3.0 Standard-B receptacles. Conversely, the USB 3.0 Standard-B receptacle can accept either a USB 2.0 or USB ;3.0 Standard-B plug.

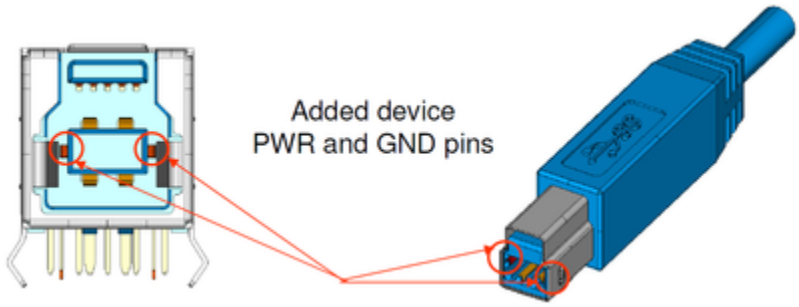


***Figure 8** : USB 3.0 Powered-B Connector*
*USB 3.0 includes a variant of the Standard-B connectors which has two additional conductors to provide power to USB adapters. Image courtesy of USB Implementers Forum*

A Powered-B variant of the Standard-B connector (Figure 8 ) is also defined by the USB 3.0 specification. The Powered-B connector has two additional pins to provide power to a USB adapter without the need for an external power supply.

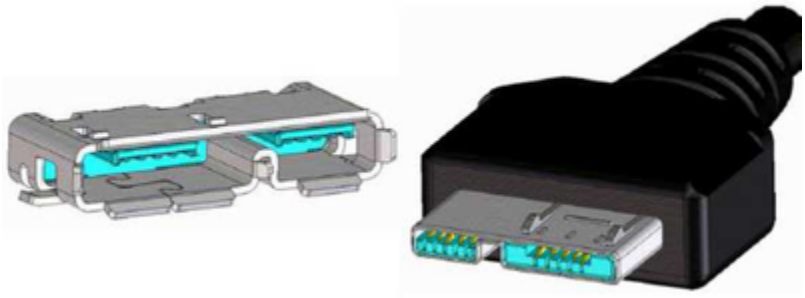

***Figure 9** : USB 3.0 Micro-A Connector*
*USB 3.0 Micro-A plug and receptacle. Image courtesy of USB Implementers Forum*

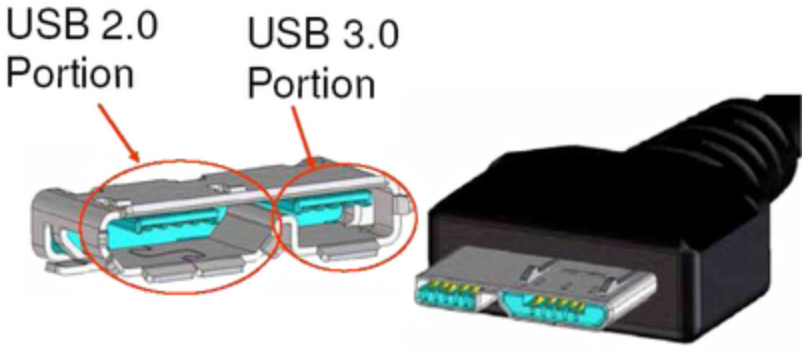


***Figure 10** : USB 3.0 Micro-B Connector*
*USB 3.0 Micro-B plug and receptacle. Image courtesy of USB Implementers Forum*

USB 3.0 also specifies Micro-A (Figure 9 ) and Micro-B (Figure 10 ) connectors. Given the small size of the original USB 2.0 micro connectors, it was not possible to add the USB 3.0 signals in the same form factor. The USB 3.0 micro plugs cannot interface with USB 2.0 receptacles, but USB 2.0 micro plugs can interface with USB 3.0 receptacles.

### USB 2.0 Signaling

All USB devices are connected by a four-wire USB cable. These four lines are , GND, and the twisted pair: D+ and D-. USB uses differential signaling on the two data lines. There are four possible digital line states that the bus can be in: single-ended zero (SE0), single-ended one (SE1), J, and K. The single-ended line states are defined the same regardless of the speed. However, the definitions of the J and K line states change depending on the bus speed. Their definitions are described in Table 1. All data is transmitted through the J and K line states. An SE1 condition should never be seen on the bus, except for allowances during transitions between the other line states.

**Table 1** : Differential Signal Encodings

| | D+ | D- |
|---|---|---|
| Single-ended zero (SE0) | 0 | 0 |
| Single-ended one (SE1) | 1 | 1 |
| Low-speed J | 0 | 1 |
| Low-speed K | 1 | 0 |
| High-/Full-speed J | 1 | 0 |
| High-/Full-speed K | 0 | 1 |

The actual data on the bus is encoded through the line states by a nonreturn-to-zero-inverted (NRZI) digital signal. In NRZI encoding, a digital 1 is represented by no change in the line state and a digital 0 is represented as a change of the line state. Thus, every

time a 0 is transmitted the line state will change from J to K, or vice versa. However, if a 1 is being sent the line state will remain the same.

USB has no synchronizing clock line between the host and device. However, the receiver can resynchronize whenever a valid transition is seen on the bus. This is possible provided that a transition in the line state is guaranteed within a fixed period of time determined by the allowable clock skew between the receiver and transmitter. To ensure that a transition is seen on the bus within the required time, USB employs bit stuffing. After 6 consecutive 1s in a data stream (i.e. no transitions on the D+ and D- lines for 6 clock periods), a 0 is inserted to force a transition of the line states. This is performed regardless of whether the next bit would have induced a transition or not. The receiver, expecting the bit stuff, automatically removes the 0 from the data stream.

### USB 3.0 Signaling

USB 3.0 signaling occurs on two dedicated pairs of differential pairs for transmission and reception. Due to the full-duplex nature of the USB 3.0 bus, the bus operates differently from a USB 2.0 bus.

USB 3.0 continues to use the concept of endpoints, pipes, and the four basic types of transfers: control, interrupt, bulk, and isochronous. USB 3.0 still uses three-part transaction of Token, Data, and Handshake, but the components are used differently. In the case of OUTs, the token is now incorporated in the data packet. In the case of INs, the token is replaced by a handshake.

There are also a number of significant changes in the USB 3.0 protocol layer to improve the efficiency of data transfers.

#### Unicast Communications

Packets are no longer broadcast on the USB bus, to allow for lower power states. In USB 2.0, packets are broadcast, consequently every device must decode the packet to determine if it needs to respond. In USB 3.0, packets are unicast, meaning that packets are sent on a directed path between the host and device as specified by routing information in the packet.

There is one exception: Isochronous Timestamp Packets (ITP) are broadcast on the bus, and provide timing information to all devices in lieu of Start of Frame packets. See Figure 20 for more information about ITP packets.

#### Asynchronous Notifications

Device polling has been eliminated in USB 3.0 to reduce bus overhead and allow for lower power states. When data is requested from a device and it is not able to respond, it can send a Not Ready packet (NRDY). When the device has freed its resources and can service the data request, it issues an Endpoint Ready packet (ERDY) informing the host that it can send another request for data.

#### Data Streaming

To improve data transfer performance, USB 3.0 introduces streams for bulk transfer endpoints. Streams are a protocol-supported method of multiplexing multiple data streams through a standard bulk pipe.

### Bus Speed

The bus speed determines the rate at which bits are sent across the bus. There are currently four speeds at which wired USB operates: low-speed (1.5 Mbps), full-speed (12 Mbps), high-speed (480 Mbps), and SuperSpeed (5 Gbps). In order to determine the bus speed of a full-speed or low-speed device, the host must simply look at the idle state of the bus. Full-speed devices have a pull-up resistor on the D+ line, whereas low-speed devices have a pull-up resistor on the D- line. Therefore, if the D+ line is high when idle, then full-speed connectivity is established. If the D- line is high when idle, then low-speed connectivity is in effect. A full-speed device does not have to be capable of running at low-speed, and vice versa. A full-speed host or hub, however, must be capable of communicating with both full-speed and low-speed devices.

With the introduction of high-speed USB, high-speed hosts and hubs must be able to communicate with devices of all speeds. Additionally, high-speed devices must be backward compatible for communication at full-speed with legacy hosts and hubs. To facilitate this, all high-speed hosts and devices initially operate at full-speed and a high-speed handshake must take place before a high-speed capable device and a high-speed capable host can begin operating at high-speed. The handshake begins when a high-speed capable host sees a full-speed device attached. Because high-speed devices must initially operate at full-speed when first connected, they must pull the D+ line high to identify as a full-speed device. The host will then issue a reset on the bus and wait to see if the device responds with a Chirp K, which identifies the device as being high-speed capable. If the host does not receive a Chirp K, it quits the high-speed handshake sequence and continues with normal full-speed operation. However, if the host receives a Chirp K, it responds to the device with alternating pairs of Chirp K's and Chirp J's to tell the device that the host is high-speed capable. Upon recognizing these alternating pairs, the device switches to high-speed operation and disconnects its pull-up resistor on the D + line. The high-speed connection is now established and both the host and the device begin communicating at high-speed. See the USB specification for more details on the high-speed handshake.

To accommodate high-speed data-rates and avoid transceiver confusion, the signaling levels of high-speed communication is much lower than that of full and low-speed devices. Full and low-speed devices operate with a logical high level of 3.3 V on the D+ and D- lines. For high-speed operation, signaling levels on the D+ and D- lines are reduced to 400 mV. Because the high-speed signaling levels are so low, full and low-speed transceivers are not capable of seeing high-speed traffic.

To accommodate the high-speed signaling levels and speeds, both hosts and devices use termination resistors. In addition, during the high-speed handshake, the device must release its full-speed pull-up resistor. But during the high-speed handshake, often times the host will activate its termination resistors before the device releases its full-speed

pull-up resistor. In these situations the host may not be able to pull the D+ line below the threshold level of its high-speed receivers. This may cause the host to see a spurious Chirp J (dubbed a Tiny J) on the lines. This is an artifact on the bus due to the voltage divider effect between the devices 1.5 Kohm pull-up resistor and the host's 45 ohm termination resistor. Hosts and devices must be robust against this situation. Once the device has switched to high-speed operation the Tiny J will no longer be present, since the device will have released its pull-up resistor.

With USB 3.0, a separate SuperSpeed USB channel co-exists in parallel with the normal USB 2.0 bus. It is important to point out that SuperSpeed USB is a full-duplex bus, thus both the host and the device act as a transmitter and receiver. In order to communicate over USB 3.0, each transmitter must detect the termination on the receiver side. If the termination is not detected, the host will downgrade its communications to USB 2.0. If the termination is detected, link training begins so that the receiver can synchronize with the transmitter. Once the link is established, the link enters U0 and data communications can begin.

### Endpoints and Pipes

The endpoint is the fundamental unit of communication in USB. All data is transferred through virtual pipes between the host and these endpoints. All communication between a USB host and a USB device is addressed to a specific endpoint on the device. Each device endpoint is a unidirectional receiver or transmitter of data; either specified as a sender or receiver of data from the host.

A pipe represents a data pathway between the host and the device. A pipe may be unidirectional (consisting of only one endpoint) or bidirectional (consisting of two endpoints in opposite directions).

A special pipe is the Default Control Pipe. It consists of both the input and output endpoints 0. It is required on all devices and must be available immediately after the device is powered. The host uses this pipe to identify the device and its endpoints and to configure the device.

Endpoints are not all the same. Endpoints specify their bandwidth requirements and the way that they transfer data. There are four transfer types for endpoints:

#### Control

Non-periodic transfers. Typically, used for device configuration, commands, and status operation.

#### Interrupt

This is a transaction that is guaranteed to occur within a certain time interval. The device will specify the time interval at which the host should check the device to see if there is new data. This is used by input devices such as mice and keyboards.

#### Isochronous

Periodic and continuous transfer for time-sensitive data. There is no error checking or retransmission of the data sent in these packets. This is used for devices that need to reserve bandwidth and have a high tolerance to errors. Examples include multimedia devices for audio and video.

**Bulk**

General transfer scheme for large amounts of data. This is for contexts where it is more important that the data is transmitted without errors than for the data to arrive in a timely manner. Bulk transfers have the lowest priority. If the bus is busy with other transfers, this transaction may be delayed. The data is guaranteed to arrive without error. If an error is detected in the CRCs, the data will be retransmitted. Examples of this type of transfer are files from a mass storage device or the output from a scanner.

### USB 2.0 Packets

All USB packets are prefaced by a SYNC field and then a Packet Identifier (PID) byte. Packets are terminated with an End-of-Packet (EOP).

The SYNC field, which is a sequence of KJ pairs followed by 2 K's on the data lines, serves as a Start of Packet (SOP) marker and is used to synchronize the devices transceiver with that of the host. This SYNC field is 8 bits long for full/low-speed and 32 bits long for high speed.

The EOP field varies depending on the bus speed. For low- or full-speed buses, the EOP consists of an SE0 for two bit times. For high-speed buses, because the bus is at SE0 when it is idle, a different method is used to indicate the end of the packet. For high-speed, the transmitter induces a bit stuff error to indicate the end of the packet. So if the line state before the EOP is J, the transmitter will send 8-bits of K. The exception to this is the high-speed SOF EOP, in which case the high-speed EOP is extended to 40-bits long. This is done for bus disconnect detection.

The PID is the first byte of valid data sent across the bus, and it encodes the packet type. The PID may be followed by anywhere from 0 to 1026 bytes, depending on the packet type. The PID byte is self-checking; in order for the PID to be valid, the last 4 bits must be a ones complement of the first 4 bits. If a received PID fails its check, the remainder of the packet will be ignored by the USB device.

There are four types of PID which are described in Table 2.

**Table 2** : USB Packet Types

| PID Type | PID Name | Description |
|---|---|---|
| Token | OUT | Host to device transfer |
| | IN | Device to Host transfer |
| | SOF | Start of Frame marker |

| | | |
|---|---|---|
| | SETUP | Host to device control transfer |
| Data | DATA0 | Data packet |
| | DATA1 | Data packet |
| | DATA2 | High-Speed Data packet |
| | MDATA | Split/High-Speed Data packet |
| Handshake | ACK | The data packet was received error free |
| | NAK | Receiver cannot accept data<br>or the transmitter could not send data |
| | STALL | Endpoint halted or control pipe request is not supported |
| | NYET | No response yet |
| Special | PRE | Preamble to full-speed hub for low-speed traffic |
| | ERR | Error handshake for Split Transaction |
| | SPLIT | Preamble to high-speed hub for low/full-speed traffic |
| | PING | High-speed flow control token |
| | EXT | Protocol extension token |

The format of the IN, OUT, and SETUP Token packets is shown in Figure 11. The format of the SOF packet is shown in Figure 12. The format of the Data packets is shown in Figure 13. Lastly, the format of the Handshake packets is shown in Figure 14.

***Figure 11** (above) : Token Packet Format*

***Figure 12** (above): Start-Of-Frame (SOF) Packet Format*

***Figure 13** (above) : Data Packet Format*

***Figure 14** (above) : Handshake Packet Format*

### Data Transactions

Data transactions occur in three phases: Token, Data, and Handshake.

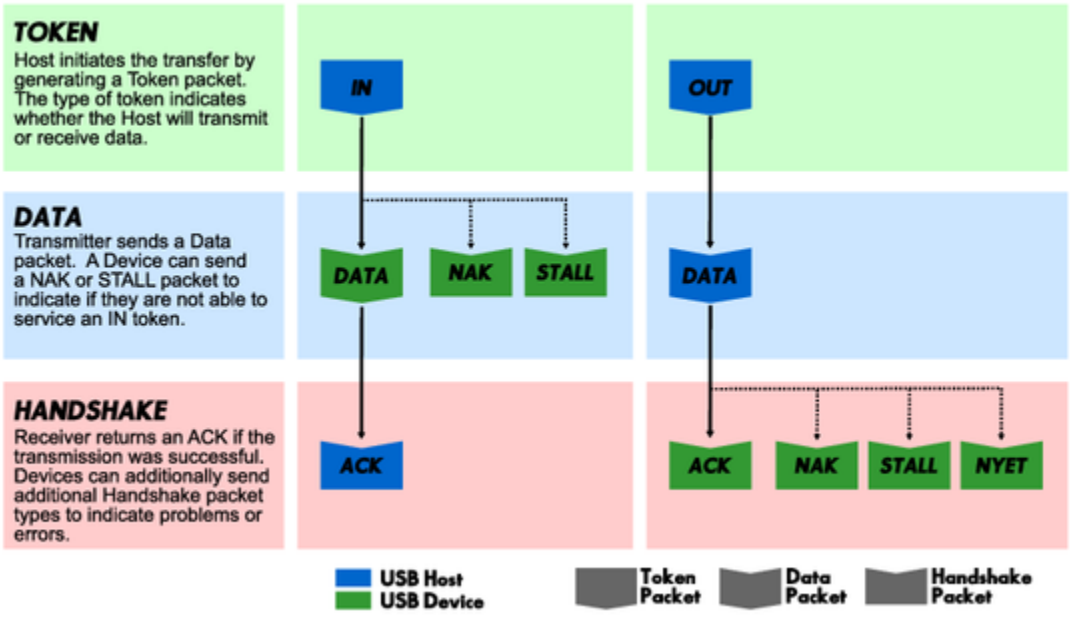


***Figure 15** : The Three Phases of a USB Transfer*

All communication on the USB is host-directed. In the Token phase, the host will generate a Token packet which will address a specific device/endpoint combination. A Token packet can be IN, OUT, or SETUP.

| | |
|---|---|
| IN | The host is requesting data from the addressed dev/ep. |
| OUT | The host is sending data to the addressed dev/ep. |
| SETUP | The host is transmitting control information to the device. |

In the data phase, the transmitter will send one data packet. For IN requests, the device may send a NAK or STALL packet during the data phase to indicate that it isnt able to service the token that it received.

Finally, in the Handshake phase the receiver can send an ACK, NAK, or STALL indicating the success or failure of the transaction.

All of the transfers described above follow this general scheme with the exception of the Isochronous transfer. In this case, no Handshake phase occurs because it is more important to stream data out in a timely fashion. It is acceptable to drop packets occasionally and there is no need to waste time by attempting to retransmit those particular packets.

**Control Transfers**

Control transfers are a group of transactions that occur on the control pipe. The control pipe is the only type of pipe which is allowed to use SETUP transactions. A control transfer consists of at least two stages called the Setup Stage and the Status Stage. Optionally, control transfers may also include a Data Stage.

The Setup Stage always consists of a single SETUP transaction. This transaction contains 8 bytes of data of which some of the bytes specify the length of the control transfer and its direction. The direction may either be host-to-device or device-to-host. If the length is not zero, then the control transfer will have a Data Stage. The Data Stage is always comprised of either IN transactions or OUT transactions depending on the direction of the control transfer. The Data Stage will never be made a mix of the two. Lastly, the Status Stage consists of an IN transaction if the control transfer was a host-to-device, or a OUT transfer was a device-to-host. The Status Stage may end in an ACK if the function completed successfully, or STALL if the function had an error. It is also possible to see a transaction STALL in the Data Stage if the device is unable to send or receive the requested data.

**Polling Transactions**

It is possible that when a host requests data or sends data that the device will not be able to service the request. This could occur if the device has no new information to provide the host or is perhaps too busy to send/receive any data. In these situations the device will NAK the host. If the data transfer is a Control or Bulk transfer, the host will

retry the transaction. However, if it is Isochronous or Interrupt transfer, it will not retry the transaction.

On a full or low-speed bus, if the transaction is repeated, it is repeated in its entirety. This is true regardless of the direction of the data transfer. If the host is requesting information, it will continue to send IN tokens until the device sends data. Until then, the device responds with a NAK, leading to the multitude of IN + NAK pairs that are commonly encountered on a bus. This does not have much consequence as an IN token is only 3 bytes and the NAK is only 1 byte. However, if the host is transmitting data there is the potential for graver consequences. For example, if the host attempted to send 64 bytes of data to a device, but the device responded with a NAK, the host will retry the entire data transaction. This requires sending the entire 64-byte data payload repeatedly until the device responds with an ACK. This has the potential to waste a significant amount of bandwidth. It is for this reason that high-speed hosts have an additional feature when the device signals the inability to accept any more data.

When a high-speed host receives a NAK after transmitting data, instead of retransmitting the entire transaction, it simply sends a 3-byte PING token to poll the device and endpoint in question. (Alternatively, if the device responds to the OUT + DATA with a NYET handshake, it means that the device accepted the data in the current transaction but is not ready to accept additional data, and the host should PING the device before transmitting more data.) The host will continue to PING the device until it responds with an ACK, which indicates to the host that it is ready to receive information. At that point, the host will transmit a packet in its entirety.

**Hub Transactions**

Hubs make it possible to expand the number of possible devices that can be attached to a single host. There are two types of hubs that are commercially available for wired USB: full-speed hubs and high-speed hubs. Both types of hubs have mechanisms for dealing with downstream devices that are not of their speed.

Full-speed hubs can, at most, transmit at 12 Mbps. This means that all high-speed devices that are plugged into a full-speed hub are automatically downgraded to full-speed data rates. On the other hand, low-speed devices are not upgraded to full-speed data rates. In order to send data to low-speed devices, the hub must actually pass slower moving data signals to those devices. The host (or high-speed hub) is the one that generates these slower moving signals on the full-speed bus. Ordinarily the low-speed ports on the hub are quiet. When a low-speed packet needs to be sent downstream, it is prefaced with a PRE PID. This opens up the low-speed ports. Note that the PRE is sent at full-speed data rates, but the following transaction is transmitted at low-speed data rates.

High-speed hubs only communicate at 480 Mbps with high-speed host. They do not downgrade the link between the host and hub to slower speeds. However, high-speed hubs must still deal with slower devices being downstream of them. High-speed hubs do not use the same mechanism as full-speed hubs. There would be a tremendous cost on bandwidth to other high-speed devices on the bus if low-speed or full-speed signaling rates were used between the host and the hub of interest. Thus, in order to save

bandwidth, high-speed hosts do not send the PRE token to high-speed hubs, but rather a SPLIT token. The SPLIT token is similar to the PRE in that it indicates to a hub that the following transaction is for a slower speed device, however the data following the SPLIT is transmitted to the hub at high-speed data rates and does not choke the high-speed bus.

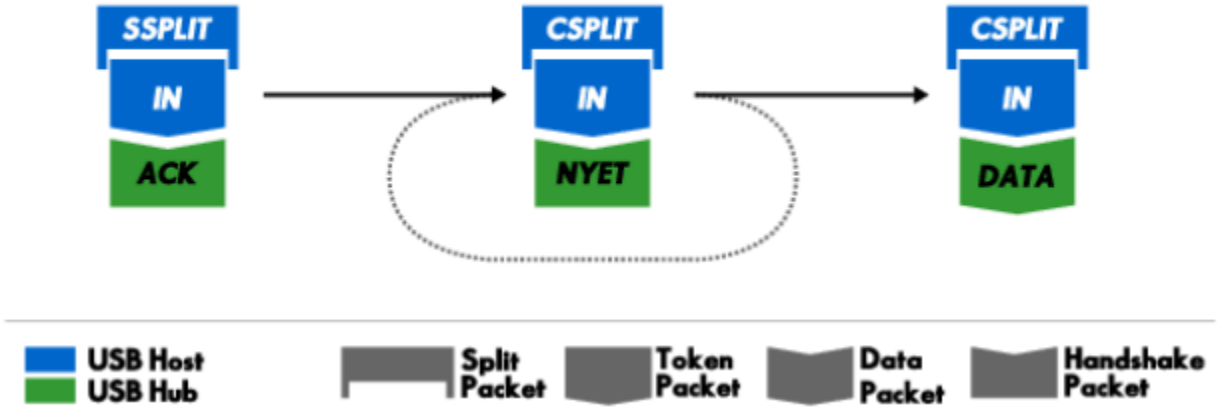


***Figure 16** : Split Bulk Transactions*
*When full/low-speed USB traffic is sent through a high-speed USB hub, the transactions are preceded by a SPLIT token to allow the hub to asynchronously handle the full/low-speed traffic without blocking other high-speed traffic from the host. In this example, bulk packets for a full-speed device are being sent through the high-speed hub. Multiple CSPLIT + IN + NYET transactions can occur on the bus until the high-speed hub is ready to return the DATA from the downstream full/low-speed device.*

Although all SPLIT transactions have the same PID, there are two over-arching types of SPLITs: Start SPLITs (SSPLIT) and Complete SPLITs (CSPLIT). SSPLITs are only used the first time that the host wishes to send a given transaction to the device. Following that, it polls the hub for the devices response with CSPLITs. The hub may respond many times with NYET before supplying the host with the devices response. Once this transaction is complete, it will begin the next hub transaction with an SSPLIT. Figure 16 illustrates an example of hub transaction.

**Start-of-Frame Transactions**

Start-of-Frame ( SOF ) transactions are issued by the host at precisely timed intervals. These tokens do not cause any device to respond, and can be used by devices for timing reasons. The SOF provides two pieces of timing information. Because of the precisely timed intervals of SOFs, when a device detects an SOF it knows that the interval

time has passed. All SOF s also include a frame number. This is an 11-bit value that is incremented on every new frame.

SOFs are also used to keep devices from going into suspend. Devices will go into suspend if they see an idle bus for an extended period of time. By providing SOF s, the host is issuing traffic on the bus and keeping devices from entering their suspended state.

Full-speed hosts will send 1 SOF every millisecond. High-speed hosts divide the frame into 8 microframes, and send an SOF at each microframe (i.e., every 125 microseconds). However, the high-speed hub will only increment the frame number after an entire frame has passed. Therefore, a high-speed host will repeat the same frame number 8 times before incrementing it.

Low-speed devices are never issued SOFs as it would require too much bandwidth on an already slower-speed bus. Instead, to keep low-speed devices from going into suspend, hosts will issue a keep-alive every millisecond. These keep-alives are short SE0 events on the bus that last for approximately 1.33 microseconds. They are not interpreted as valid data, and have no associated PID.

**Extended Token Transactions**

The new Link Power Management addendum to the USB 2.0 Specification has expanded the number of PIDs through the use of the previously reserved PID, 0xF0. The extended token format is a two phase transaction that begins with a standard token packet that has the EXT PID. Following this packet is the extended token packet, which takes a similar form. It begins with an 8-bit SubPID and ends with a 5-bit CRC, however the 11 remaining bits in the middle will have different meaning depending on the type of SubPID.

***Figure 17** (above) : Extended Token Transaction*
*In an extended token transaction, the token phase of the transaction has two token packets. The first packet uses the EXT PID. The content of the second packet will depend on the particular SubPID specification. The subsequent Data and Handshake phases will depend on the value of the SubPID as well.*

Following this token phase, the device will respond with the appropriate data or handshake, depending on the protocol associated with that SubPID. Currently, the only defined SubPID is for link power management ( LPM ). For more details, please refer to the Link Power Management addendum.

## USB 3.0 Packets

USB 3.0 supports the same types of data transfers: control, interrupt, bulk, and isochronous. However the packet structure has changed to support the new features in USB 3.0.

### USB 3.0 General Packet Structures

Packets in USB 3.0 generally come in 3 different patterns.

#### Header Packet

Header Packets consist of three parts: header packet framing, packet header, and a link control word. Note that "SHP" is a a K-symbol which stands for "start header packet". The header is protected by CRC-16, and the link control word is protected by CRC-5.

#### Data Payload Packet

Data Payload Packets send application data and are protected by CRC-32. Note that "SDP" stands for "start data packet payload".

#### Link Command Packet

Link Command Packets are used to control various link-specific features, including low power states and flow control. A Link Command Packet atually consists of two identical Link Command Words, where each Link Command Word is protected by a CRC-5. Note that "SLC" stands for "start link command".

#### Link Management Packets (LMP)

Link Management Packets (LMP) are a type of header packet used to manage the link between two ports (Figure 18 ). Because these packets are used to manage a single link, they only travel between the two link partners and therefore require no addressing information.

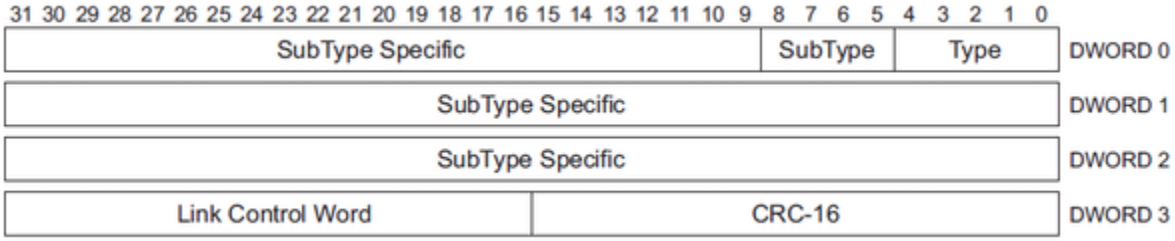

***Figure 18** (above): Link Management Packet (LMP)*
*Link Management Packets are used to manage the link between two link partners. Image courtesy of USB Implementers Forum.*

The LMP commands to manage a link are listed below. Please consult the USB 3.0 specifications for more details.

- Set Link Function
- U2 Inactivity Timeout
- Vendor Device Test
- Port Capability
- Port Configuration
- Port Configuration Response

**Transaction Packets (TP)**

Transaction Packets (TP) are a type of header packet used to control the flow of data packets end-to-end between the host and device (Figure 19 ). Since these packets may traverse a number of links, each TP has a route string which is used by hubs to route the packet directly to the intended device.

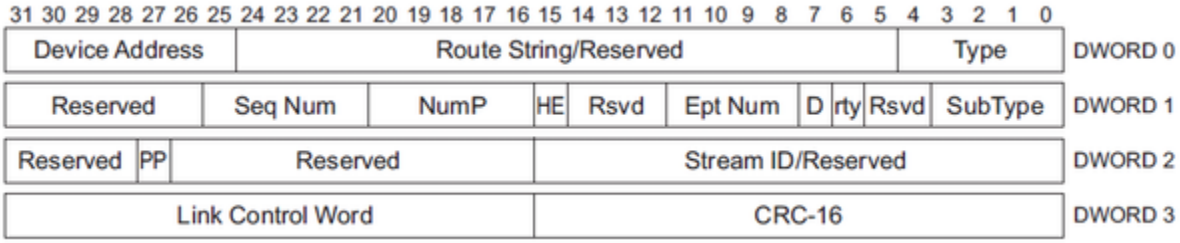


***Figure 19** (above) : Transaction Packet (TP)*
*Transaction Packets are used to control the flow of packets between the host and device. Image courtesy of USB Implementers Forum.*

TPs do not contain application data and have a number of different subtypes:

- Acknowledgement (ACK)
- Not Ready (NRDY)

- Endpoint Ready (ERDY)
- STATUS
- STALL
- Device Notification (DEV_NOTIFICATION)
- PING
- PING_RESPONSE

**Data Packets (DP)**

Data Packets (DP) are used to transmit application data and are comprised of two parts: a data packet header (DPH) and a data packet payload (DPP) (Figure 20 ).

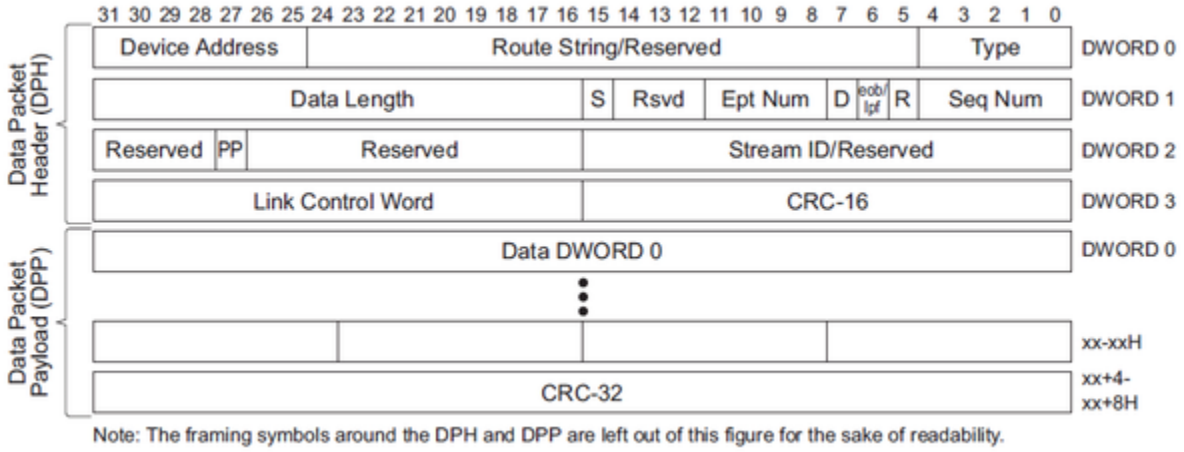


***Figure 20** : Data Packet (DP)*
*Data Packets are used to transmit application data between the host and device. A Data Packet is composed of a Data Packet Header (DPH) and the Data Packet Payload (DPP). Image courtesy of USB Implementers Forum.*

Since data is being sent between the host and device, DP packets have a route string to direct it to intended device.

**Isochronous Timestamp Packets (ITP)**

Isochronous Timestamp Packets (ITP) are used to send timestamps to all devices for synchronization (Figure 20 ). ITPs are the only packets that are broadcast by the host to all active devices. Since this packet is broadcast, it does not require a route string.

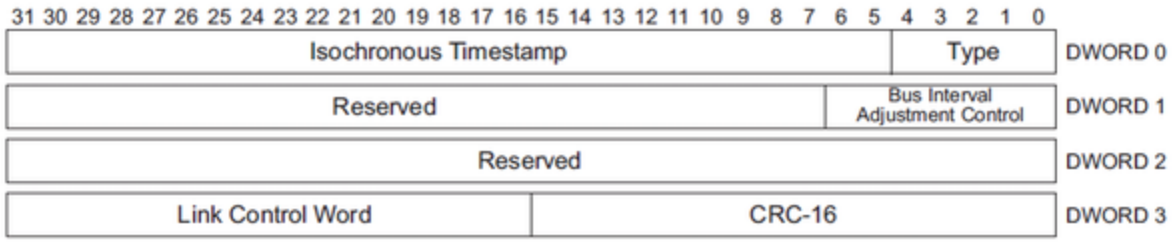


***Figure 20** : Isochronous Timestamp Packet (LMP)*
*Link Management Packets send timestamps to active devices which is used for synchronization. Image courtesy of USB Implementers Forum.*

Only hosts are allowed to send ITPs, and only when the host port is already in the U0 state. Devices are not required to respond to the ITP.

### Enumeration and Descriptors

When a device is plugged into a host PC, the device undergoes Enumeration. This means that the host recognizes the presence of the device and assigns it a unique 7-bit device address. The host PC then queries the device for its descriptors, which contains information about the specific device. There are various types of descriptors as outlined below.

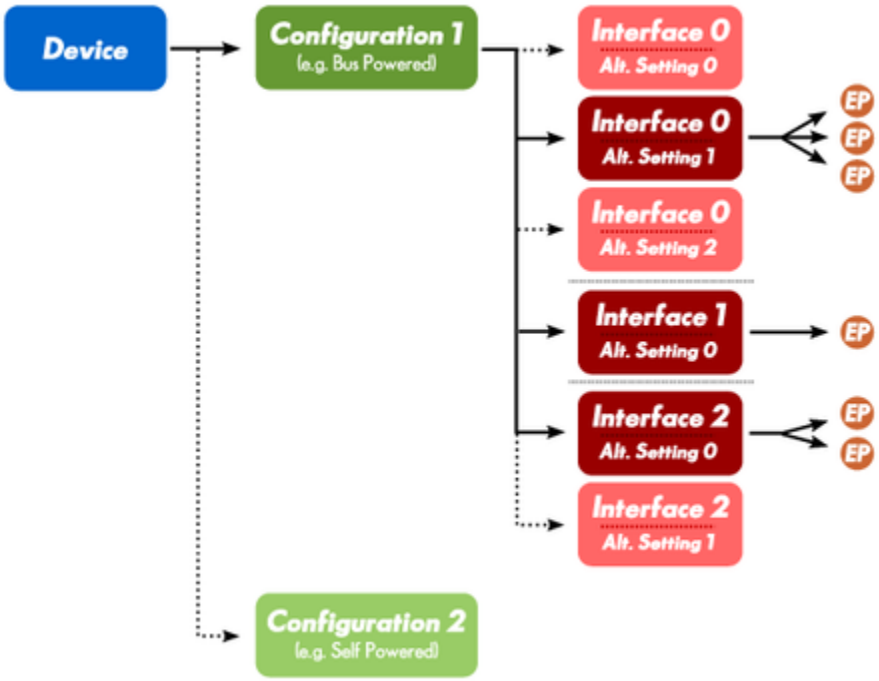

***Figure 22** : USB Descriptors*
*Hierarchy of descriptors of a USB device. A device has a single Device descriptor. The Device descriptor can have multiple Configuration descriptors, but only a single one can be active at a time. The Configuration descriptor can define one or more Interface descriptors. Each of the Interface descriptors can have one or more alternate settings, but only one setting can be active at a time. The Interface descriptor defines one or more Endpoints.*

- *Device Descriptor* : Each USB device can only have a single Device Descriptor. This descriptor contains information that applies globally to the device, such as serial number, vendor ID, product ID, etc. The device descriptor also has information about the device class. The host PC can use this information to help determine what driver to load for the device.
- *Configuration Descriptor* : A device descriptor can have one or more configuration descriptors. Each of these descriptors defines how the device is powered (e.g. bus-powered or self-powered), the maximum power consumption, and what interfaces are available in this particular setup. The host can choose whether to read just the configuration descriptor or the entire hierarchy (configuration, interfaces, and alternate interfaces) at once.
- *Interface Descriptor* : A configuration descriptor defines one or more interface descriptors. Each interface number can be subdivided into multiple alternate interfaces that help more finely modify the characteristics of a device. The host PC selects particular alternate interface depending on what functions it wishes to access. The interface also has class information which the host PC can use to determine what driver to use.
- *Endpoint Descriptor* : An interface descriptor defines one or more endpoints. The endpoint descriptor is the last leaf in the configuration hierarchy and it defines the bandwidth requirements, transfer type, and transfer direction of an endpoint. For transfer direction, an endpoint is either a source (IN) or sink (OUT) of the USB device.
- *String Descriptor* : Some of the configuration descriptors mentioned above can include a string descriptor index number. The host PC can then request the unicode encoded string for a specified index. This provides the host with human

readable information about the device, including strings for manufacturer name, product name, and serial number.

### Device Class

USB devices vary greatly in terms of function and communication requirements. Some devices are single-purpose, such as a mouse or keyboard. Other devices may have multiple functionalities that are accessible via USB such as a printer/scanner/fax device.

The USB-IF Device Working Group defines a discreet number of device classes. The idea was to simplify software development by specifying a minimum set of functionality and characteristics that is shared by a group of devices and interfaces. Devices of the same class can all use the same USB driver. This greatly simplifies the use of USB devices and saves the end-user the time and hassle of installing a driver for every single USB device that is connected to their host PC.

For example, input devices such as mice, keyboards and joysticks are all part of the HID (Human Interface Device) class. Another example is the Mass Storage class which covers removable hard drives and keychain flash disks. All of these devices use the same Mass Storage driver which simplifies their use.

However, a device does not necessarily need to belong to a specific device class. In these cases, the USB device will require its own USB driver that the host PC must load to make the functionality available to the host.

### On-The-Go (OTG)

The OTG supplement to the USB 2.0 spec provides methods for mobile devices to communicate with each other, actively switch the role of host and device, and also request sessions from each other when power to the USB is removed.

The initial role of host and device is determined entirely by the USB connector itself. All OTG capable peripherals will have a 5-pin Micro-AB receptacle which can receive either the Micro-A or Micro-B plug. If the peripheral receives the Micro-A plug, then it behaves as the host. If it receives the Micro-B plug, then it behaves as the device. However, there may be certain situations where a peripheral received the Micro-B plug, but needs to behave as the host. Rather than request that the user swap the cable orientation, the two peripherals have the ability to swap the roles of host and device through the Host Negotiation Protocol (HNP).

The HNP begins when the A-device finishes using the bus and stops all bus activity. The B-device detects this and will release its pull-up resistor. When the A-device detects the SE0, it responds by activating its pull-up. Once the B-device detects this condition, the B-device issues reset and begins standard USB communication as the host.

In order to conserve power, A-devices are allowed to stop providing power to the USB. However, there could be situations where the B-device wants to use the bus and $V_{BUS}$ is turned off. It is for this reason that the OTG supplement describes a method for allowing

the B-device to request a session from the A-device. Upon successful completion of the Session Request Protocol (SRP), the A-device will power the bus and continue standard USB transactions.

The SRP is broken up into two stages. From a disconnected state, the B-device must begin an SRP by driving one of its data lines high for a sufficient duration. This is called data-line pulsing. If the A-device does not respond to this, the B-device will drive the $V_{BUS}$ above a specified threshold and release it, thereby completing $V_{BUS}$ pulsing. If the A-device still does not begin a session, the B-device may start the SRP over again, provided the correct initial conditions are met.

For more details on OTG, please see the *On-The-Go Supplement to the USB 2.0 Specification* .

### 1.1.4 References

- USB Implementers Forum

## 1.2 I²C Background

### 1.2.1 I²C History

When connecting multiple devices to a microcontroller, the address and data lines of each devices were conventionally connected individually. This would take up precious pins on the microcontroller, result in a lot of traces on the PCB, and require more components to connect everything together. This made these systems expensive to produce and susceptible to interference and noise.

To solve this problem, Philips developed Inter-IC bus, or I²C, in the 1980s. I²C is a low-bandwidth, short distance protocol for on board communications. All devices are connected through two wires: serial data (SDA) and serial clock (SCL).

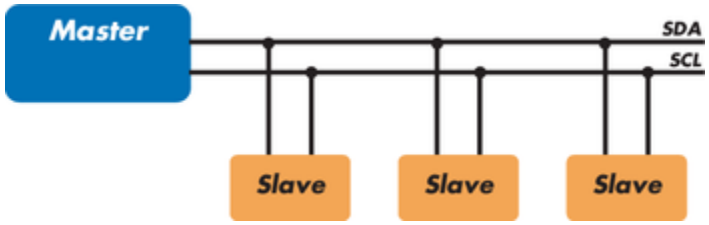

***Figure 23** : Sample I²C Implementation.*
*Regardless of how many slave units are attached to the I²C bus, there are only two signals connected to all of them. Consequently, there is additional overhead because an addressing mechanism is required for the master device to communicate with a specific slave device.*

Because all commnication takes place on only two wires, all devices must have a unique address to identify it on the bus. Slave devices have a predefined address, but the lower bits of the address can be assigned to allow for multiples of the same devices on the bus.

### 1.2.2 I²C Theory of Operation

I²C has a master/slave protocol. The master initiates the communication. Here is a simplified description of the protocol. For precise details, please refer to the Philips I²C specification. The sequence of events are as follows:

1. The master device issues a start condition. This condition informs all the slave devices to listen on the serial data line for their respective address.
2. The master device sends the address of the target slave device and a read/write flag.
3. The slave device with the matching address responds with an acknowledgment signal.
4. Communication proceeds between the master and the slave on the data bus. Both the master and slave can receive or transmit data depending on whether the communication is a read or write. The transmitter sends 8 bits of data to the receiver, which replies with a 1-bit acknowledgment.
5. When the communication is complete, the master issues a stop condition indicating that everything is done.

Figure 24 shows a sample bitstream of the I²C protocol.

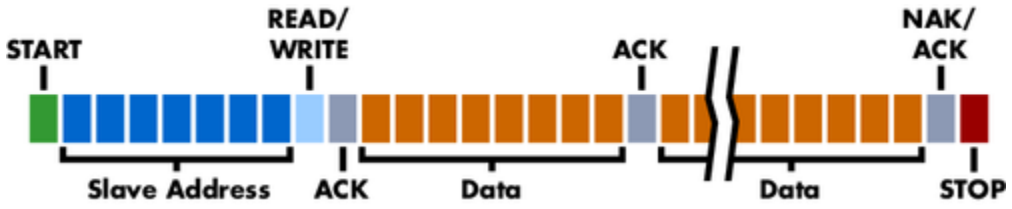

***Figure 24** : I²C Protocol.*
*Since there are only two wires, this protocol includes the extra overhead of the addressing and acknowledgement mechanisms.*

### 1.2.3 I²C Features

I²C has many features other important features worth mentioning. It supports multiple data speeds: standard (100 kbps), fast (400 kbps) and high-speed (3.4 Mbps) communications.

Other features include:

- Built-in collision detection
- 10-bit Addressing
- Multi-master support
- Data broadcast (general call)

For more information about other features, see the references at the end of this section.

### 1.2.4 I²C Benefits and Drawbacks

Since only two wires are required, I²C is well suited for boards with many devices connected on the bus. This helps reduce the cost and complexity of the circuit as additional devices are added to the system.

Due to the presence of only two wires, there is additional complexity in handling the overhead of addressing and acknowledgments. This can be inefficient in simple configurations and a direct-link interface such as SPI might be preferred.

### 1.2.5 I²C References

- I²C bus – *NXP (Philips) Semiconductors Official I²C website*
- I²C (Inter-Integrated Circuit) Bus Technical Overview and Frequently Asked Questions – *Embedded Systems Academy*
- Introduction to I²C – *Embedded.com*
- I²C – *Open Directory Project Listing*

# 1.3 SPI Background

## 1.3.1 SPI History

SPI is a serial communication bus developed by Motorola. It is a full-duplex protocol which functions on a master-slave paradigm that is ideally suited to data streaming applications.

## 1.3.2 SPI Theory of Operation

SPI requires four signals: clock (SCLK), master output/slave input (MOSI), master input/slave output (MISO), and slave select (SS).

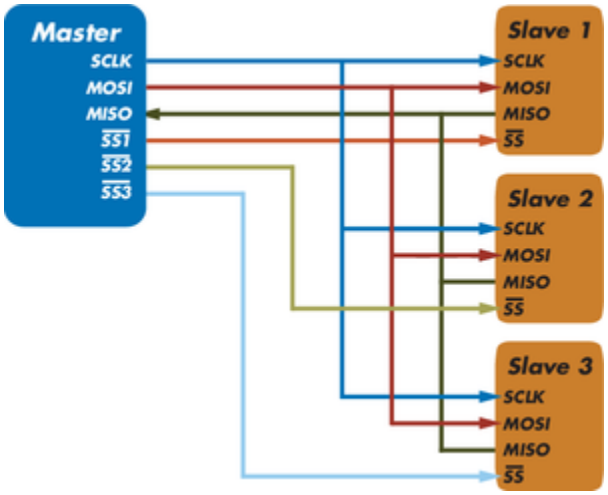


***Figure 25** : Sample SPI Implementation.*
*Each slave device requires a separate slave select signal (SS). This means that as devices are added, the circuit increases in complexity.*

Three signals are shared by all devices on the SPI bus: SCLK, MOSI, and MISO. SCLK is generated by the master device and is used for synchronization. MOSI and MISO are the data lines. The direction of transfer is indicated by their names. Data is always transferred in both directions in SPI, but an SPI device interested in only transmitting data can choose to ignore the receive bytes. Likewise, a device only interested in the incoming bytes can transmit dummy bytes.

Each device has its own SS line. The master pulls low on a slave's SS line to select a device for communication.

The exchange itself has no pre-defined protocol. This makes it ideal for data-streaming applications. Data can be transferred at high speed, often into the range of the tens of

megahertz. The flipside is that there is no acknowledgment, no flow control, and the master may not even be aware of the slave's presence.

### 1.3.3 SPI Modes

Although there is no protocol, the master and slave need to agree about the data frame for the exchange. The data frame is described by two parameters: clock polarity (CPOL) and clock phase (CPHA). Both parameters have two states which results in four possible combinations. These combinations are shown in Figure 26.

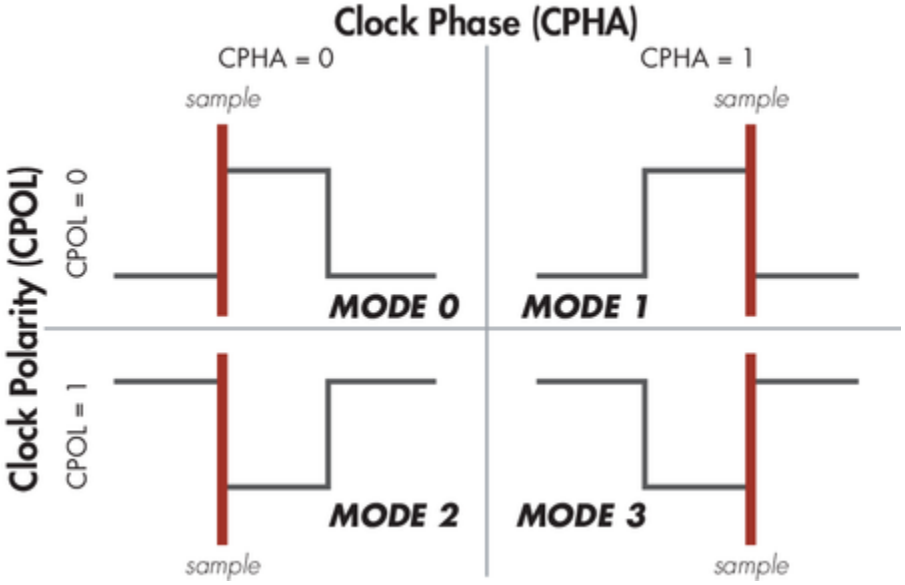


***Figure 26** : SPI Modes*
*The frame of the data exchange is described by two parameters, the clock polarity (CPOL) and the clock phase (CPHA). This diagram shows the four possible states for these parameters and the corresponding mode in SPI.*

### 1.3.4 SPI Benefits and Drawbacks

SPI is a very simple communication protocol. It does not have a specific high-level protocol which means that there is almost no overhead. Data can be shifted at very high rates in full duplex. This makes it very simple and efficient in a single-master single-slave scenario.

Because each slave needs its own SS, the number of traces required is n+3, where n is the number of SPI devices. This means increased board complexity when the number of slaves is increased.

### 1.3.5 SPI References

- Introduction to Serial Peripheral Interface – *Embedded.com*
- SPI – Serial Peripheral Interface

## 1.4 MDIO Background

### 1.4.1 MDIO History

Management Data Input/Output, or MDIO, is a 2-wire serial bus that is used to manage PHYs or physical layer devices in media access controllers (MACs) in Gigabit Ethernet equipment. The management of these PHYs is based on the access and modification of their various registers.

MDIO was originally defined in Clause 22 of IEEE RFC802.3. In the original specification, a single MDIO interface is able to access up to 32 registers in 32 different PHY devices. These registers provide status and control information such as: link status, speed ability and selection, power down for low power consumption, duplex mode (full or half), auto-negotiation, fault signaling, and loopback.

To meet the needs the expanding needs of 10-Gigabit Ethernet devices, Clause 45 of the 802.3ae specification provided the following additions to MDIO:

- Ability to access 65,536 registers in 32 different devices on 32 different ports
- Additional OP-code and ST-code for Indirect Address register access for 10 Gigabit Ethernet
- End-to-end fault signaling
- Multiple loopback points
- Low voltage electrical specification

### 1.4.2 MDIO Theory of Operation

The MDIO bus has two signals: Management Data Clock (MDC) and Managment Data Input/Ouput (MDIO).

MDIO has specific terminology to define the various devices on the bus. The device driving the MDIO bus is identified as the Station Management Entity (STA). The target devices that are being managed by the MDC are referred to as MDIO Manageable Devices (MMD).

The STA initiates all communication in MDIO and is responsible for driving the clock on MDC. MDC is specified to have a frequency of up to 2.5 MHz.

### 1.4.3 Clause 22

Clause 22 defines the MDIO communication basic frame format (Figure 27 ) which is composed of the following elements:

***Figure 27** : Basic MDIO Frame Format*

**Table 3** : Clause 22 format

| | | |
|---|---|---|
| ST | 2 bits | Start of Frame (01 for Clause 22) |
| OP | 2 bits | OP Code |
| PHYADR | 5 bits | PHY Address |
| REGADR | 5 bits | Register Address |
| TA | 2 bits | Turnaround time to change bus ownership from STA to MMD if required |
| DATA | 16 bits | Data<br>Driven by STA during write<br>Driven by MMD during read |

The frame format only allows a 5-bit number for both the PHY address and the register address, which limits the number of MMDs that the STA can interface. Additionally, Clause 22 MDIO only supports 5 V tolerant devices and does not have a low voltage option.

### 1.4.4 Clause 45

In order to address the deficiencies of Clause 22, Clause 45 was added to the 802.3 specification. Clause 45 added support for low voltage devices down to 1.2 V and extended the frame format (Figure 28 ) to provide access to many more devices and registers. Some of the elements of the extended frame are similar to the basic data frame:

***Figure 28** : Extended MDIO Frame Format*

**Table 4** : Clause 45 format

| | | |
|---|---|---|
| ST | 2 bits | Start of Frame (00 for Clause 45) |
| OP | 2 bits | OP Code |
| PHYADR | 5 bits | PHY Address |
| DEVTYPE | 5 bits | Device Type |
| TA | 2 bits | Turnaround time to change bus ownership from STA to MMD if required |
| ADDR/DATA | 16 bits | Address or Data<br>Driven by STA for address<br>Driven by STA during write<br>Driven by MMD during read<br>Driven by MMD during read-increment-address |

The primary change in Clause 45 is how the registers are accessed. In Clauses 22, a single frame specified both the address and the data to read or write. Clause 45 changes this paradigm. First an address frame is sent to specify the MMD and register. A second frame is then sent to perform the read or write.

The benefits of adding this two cycle access are that Clause 45 is backwards compatible with Clause 2, allowing devices to interoperate with each other. Secondly, by creating a address frame, the register address space is increased from 5 bits to 16 bits, which allows an STA to access 65,536 different registers.

In order to accomplish this, several changes were made in the composition of the data frame. A new ST code (00) is defined to identify Clause 45 data frames. The OP codes were expanded to specify an address frame, a write frame, a read frame, or a read and post read increment address frame. Since the register address is no longer needed, this field is replaced with DEVTYPE to specify the targeted device type. The expanded device type allows the STA to access other devices in addition to PHYs.

Additional details about Clause 45 can be found on the IEEE 802.3 workgroup website.

### 1.4.5 MDIO References

- IEEE 802 LAN/MAN Standards Committee
- Use The MDIO Bus To Interrogate Complex Devices – *Electronic Design Magazine*

# 2 Hardware Specifications

## 2.1 Beagle USB 5000 SuperSpeed Protocol Analyzer v2

### 2.1.1 Front Panel

The front panel (Figure 29 ) of the Beagle USB 5000 Protocol Analyzer offers a number of LED indicators and connectors.

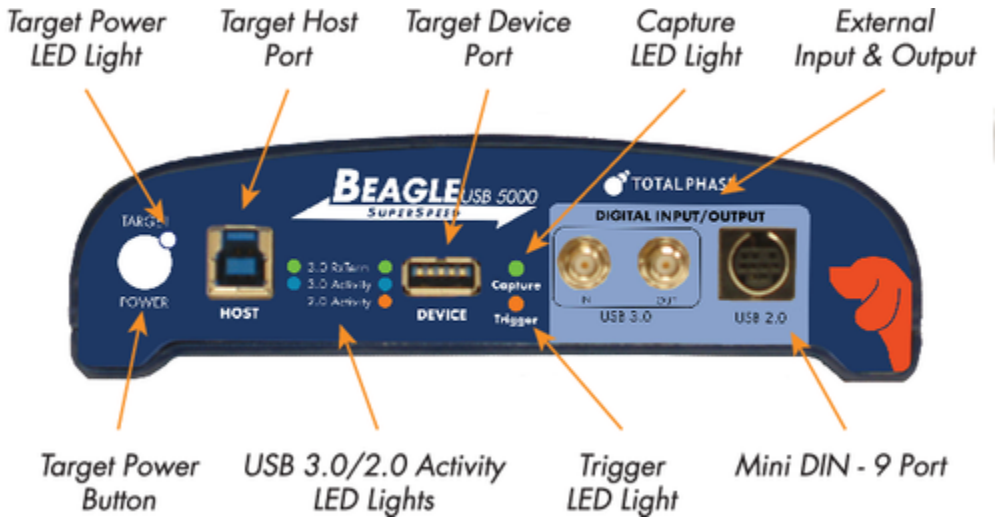


***Figure 29** : Beagle USB 5000 SuperSpeed Protocol Analyzer v2 - Front*

#### Analyzer Power

The Beagle USB 5000 analyzer power indicator is integrated into the Total Phase logo located above the USB 3.0 External Input/Output connectors. When the analyzer is powered, the large circle in the Total Phase logo will be illuminated.

#### Target Power

The Target Power indicator consists of two elements: the large white circular button and the LED indicator in the upper right corner of the button. When the button is pressed, $V_{BUS}$ will be disconnected between the target host and target device.

When $V_{BUS}$ is present, the white LED will be on. When the button is pressed to disconnect $V_{BUS}$, the white LED will turn off. $V_{BUS}$ can also be disconnected by software. Should $V_{BUS}$ be disconnected in this way, the LED will turn off as expected.

### Target Host and Target Device Ports

The Target Host port is a SuperSpeed USB A receptacle. While this receptacle may appear to be a USB 2.0 port, this receptacle features the 5 extra conductors required for SuperSpeed USB. This receptacle is compatible with both USB 2.0 and USB 3.0 cables. However, in order to monitor USB 3.0 traffic, a USB 3.0 cable must be used to connect to a USB 3.0 host.

The Target Device port is a SuperSpeed USB B receptacle. This receptacle is compatible with both USB 2.0 and USB 3.0 cables. However, in order to monitor USB 3.0 traffic, a USB 3.0 cable must be used to connect to a USB 3.0 device.

The Beagle USB analyzer must be powered to ensure that the USB 3.0 ports function properly. Failure to power the Beagle USB analyzer before attaching the target USB host and device may result in unexpected behavior.

As long as the analyzer is powered on, the USB 3.0 connectors will be active and will transmit USB 3.0 data, even if the analyzer is not actively capturing data. This is true even if the analyzer is not connected to the Analysis computer.

### Activity Indicators

Between the Target Host and Target Device ports, there are a number of LED indicators.

#### RxTerm

RxTerm, or receiver termination, indicators are illuminated when the presence of the USB 3.0 termination resistor is detected.

During normal operation, it is possible that the receiver termination indicator for the Target Device may remain illuminated even though the device may have been removed from the analyzer. If the target host continues to send data, regardless of the presence of the target device, the analyzer will assume that the device is still connected to the bus.

A sophisticated algorithm is used to balance the detection of the termination status of the line and maintaining data capture fidelity. In situations where there is a conflict, the analyzer will focus on maintaining the data capture at the expense of a delayed receiver termination detection.

For more detailed information about receiver termination detection, please refer to Section 3.3.1 in the Device Operation Section.

#### USB 3.0 Activity

The USB 3.0 Activity LEDs are illuminated when there is USB 3.0 bus activity and a data capture is active. The LED blink speed is proportional to the amount of USB 3.0 traffic on

the bus. If the analyzer is not capturing data, the LEDs will not be active even if there is USB 3.0 traffic on the bus.

Please note that there is a minimum activity threshold to activate the LEDs. In general, the LEDs will only be active in the U0 state. Periodic link commands or LFPS traffic may not necessarily be sufficient to cross the activity threshold.

#### USB 2.0 Activity

The USB 2.0 Activity LED is illuminated when there is USB 2.0 bus activity and the data capture is active. The LED blink speed is proportional to the amount of USB 2.0 traffic on the bus. If the analyzer is not capturing data, the LEDs will not be active even if there is USB 2.0 traffic on the bus.

Please note that, unlike the Beagle USB 480 analyzer, the LED will not be illuminated if there is no USB 2.0 activity.

#### Capture

The Capture LED indicator will be illuminated when a capture is active. Once the capture has ended, the Capture indicator will continue to blink while data is being transferred to the Analysis computer. The Capture LED will turn off once the data transfer is complete.

#### Trigger

The Trigger LED indicator will be illuminated once the trigger occurs. The indicator will remain active until all the data has been downloaded to the Analysis PC.

### External Inputs and Outputs

The Beagle USB 5000 analyzer features two separate sets of external inputs and outputs.

#### USB 3.0 Input and Output

The USB 3.0 input and output are the two SMA connectors located on the front panels. Both the input and the output have an impedance of 50 ohms and are rated for 1.8 V, 12 mA.

**WARNING: The USB 3.0 Digital Input and Output are only rated for 1.8 V.** The USB 3.0 input and output of the Beagle USB 5000 analyzer have been optimized for maximum edge performance at 125 MHz. **Applying signals with higher voltage will damage your analyzer and is not covered by the warranty.**

The external USB 3.0 input has a latency of 0 to 25 ns from when the input is asserted to when the analyzer detects the assertion. This input can be used as an external capture trigger or as a way to synchronize USB 3.0 traffic with external logic. It is possible to capture a 125 MHz signal pulse with the external input, however if these events are too

frequent, the analyzer will throttle the external input signal in order to maintain capture fidelity.

The USB 3.0 external output allows users to output USB 3.0 events to external devices such as a oscilloscope or logic analyzer. The output has a short latency of 50 to 75 ns from when an event occurs to when the output is asserted on the external output SMA connector. Please see Section 3.3.5 for more information about External Output behavior.

**USB 2.0 Inputs and Outputs**

The USB 2.0 External Inputs and Outputs are available through the Mini-DIN9 port. The output level is 3 V and the input is 3.3 V tolerant. The pin out and functionality of this connector is the same as the Beagle USB 480 analyzer which is described in Section 2.2.1.

**WARNING: The USB 2.0 Inputs and Outputs are only rated for 3.3 V. Applying signals with higher voltage will damage your analyzer and is not covered by the warranty.**

### 2.1.2 Back Panel

The back panel (Figure 30 ) of the Beagle USB 5000 Protocol Analyzer provides the power connector and downlink connector to the Analysis PC.

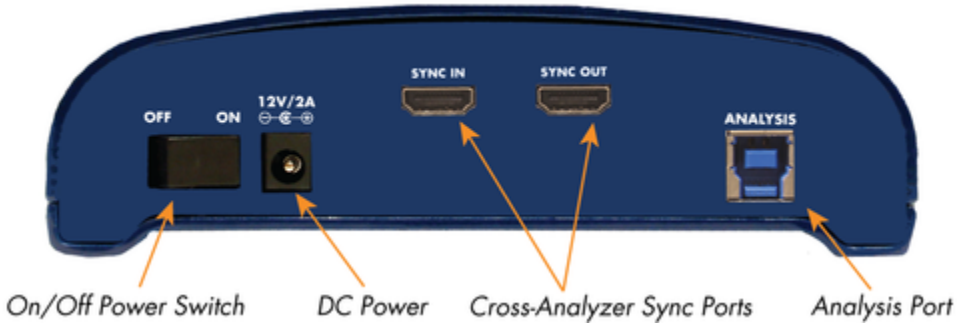


***Figure 30** : Beagle USB 5000 SuperSpeed Protocol Analyzer v2 - Back*

**Analysis**

The Analysis port is a SuperSpeed USB downlink. The Beagle analyzer must be connected with a standard USB 3.0 or USB 2.0 cable to the Analysis computer.

#### Power

The Beagle USB 5000 analyzer includes a 36 W AC power adapter. To ensure the proper operation of the Beagle analyzer it must be powered on before any devices are connected to the analyzer.

The DC connector has positive polarity and has a barrel plug with dimensions of 5.5 mm x 3.5 mm x 9.5 mm.

#### HDMI Ports

The Beagle USB 5000 analyzer has two HDMI ports, labeled SYNC OUT and SYNC IN, on the back panel. These HDMI ports are provided to allow capture synchronization between two or more Beagle USB 5000 analyzers. Do not connect the HDMI ports to any arbitrary HDMI-compatible device. The HDMI ports on a Beagle USB 5000 analyzer back panel should only be connected to the HDMI ports on another Beagle USB 5000 analyzer.

For details on using the Cross-Analyzer Sync feature, refer to Section 3.3.7.

**Note** : Cross-Analyzer Sync HDMI ports are only available on hardware v2.00 or later.

### 2.1.3 On-board Buffer

The Beagle USB 5000 analyzer includes a 2  B USB 3.0 memory buffer. This memory buffer can be upgraded to 4 GB with an optional upgrade package. The Beagle USB 5000 analyzer has a parallel 128 MB USB 2.0 buffer which is used for USB 2.0 only captures and simultaneous USB 2.0/3.0 captures.

The memory provides a temporary FIFO storage buffer for capture data. This data is constantly streamed from the analyzer to the Analysis computer over the high-speed data downlink after the trigger condition has been met. Consequently, the memory buffer is constantly being emptied, which frees up resources for additional data. This means that the Beagle analyzer is capable of capturing significantly more data than the available on-board hardware buffer.

### 2.1.4 Active Analog Buffer

The Beagle USB 5000 analyzer features an active analog buffer circuit as part of the capture front end of the SuperSpeed signals to provide optimal signal integrity. Each signal transmitted, by the host and device, is buffered and retransmitted. The signal is **not retimed** to the respective receiver.

At the same time as data is retransmitted to the target receiver, a parallel signal is passed to the analyzer for analysis. The maximum latency for the analog buffering is less than 1 ns and thus the circuitry is non-intrusive from the perspective of the host and

device. Due to the high-speed signaling of USB 3.0 data, it is not practical to passively tap the data lines between host and device outside of very high-end oscilloscopes and bit error rate testers.

#### Configurable SuperSpeed Front-End

For the convenience of the user, it is possible to modify the receiver and transmitter settings of the active buffer circuitry.

On the receiver side, users are able to modify the receiver equalization settings, though often this is not necessary.

On the transmitter side, users are able to adjust the signal level of the output. By configuring the levels sent by the transmitter, it is possible to test the sensitivity of the receiver of the USB 3.0 device. The characteristics of the transmitter can also be modified by changing the output pre-emphasis.

### 2.1.5 Signal Specifications and Power Consumption

#### Speed

The Beagle USB 5000 Protocol Analyzer supports capture of all wired USB speeds. The analyzer has automatic speed detection as well as manual speed locking.

#### ESD Protection

The Beagle analyzer has built-in electrostatic discharge protection to prevent damage to the unit from high voltage static electricity.

#### Power consumption

When the Beagle analyzer is connected, it consumes a maximum of approximately 2.5 mA from the capture host.

## 2.2 Beagle USB 480 Protocol Analyzer

The Beagle USB 480 series of protocol analyzers consist of the following:

- Beagle USB 480 Protocol Analyzer
- Beagle USB 480 Power Protocol Analyzer, Standard Edition
- Beagle USB 480 Power Protocol Analyzer, Ultimate Edition

Beagle USB 480 Power Protocol Analyzers feature larger on=board memory buffers and $V_{BUS}$ Current/Voltage Monitoring. The Ultimate Edition includes Advanced USB 2.0 Match/Action Triggers and Filters. All Beagle USB 480 Protocol Analyzers have identical hardware interface.

### 2.2.1 Connector Specification

On one side of the Beagle USB 480 monitor is a single USB-B receptacle. This is the **Analysis** side (Figure 31 ). This port connects to the analysis computer that is running the Beagle Data Center software or custom application. Furthermore, the Beagle USB 480 analyzer **Analysis** side **must be plugged in at any time a target device is plugged in**. This is to ensure that all connections are properly powered.

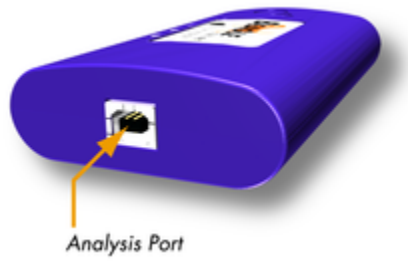


***Figure 31** : Beagle USB 480 Protocol Analyzer - Analysis Side*

The opposite side is the **Capture** side (Figure 32 ), and it contains a USB-A and USB-B receptacle. These are used to connect the target host computer to the target device. The target host computer can be the same computer as the analysis computer, although it may not be optimal under certain conditions.

***Figure 32** : Beagle USB 480 Protocol Analyzer - Capture Side*

The **Capture** side acts as a USB pass-through. In order to remain within the USB 2.0 specifications, no more than 5 meters of USB cable should be used in total between the target host computer and the target device.

The **Capture** side also includes a mini-DIN 9 connector which serves as a connection to the digital inputs and outputs. Its pin outs are described in Figure 33 and the cable coloring for the included cable are described in Table 5.

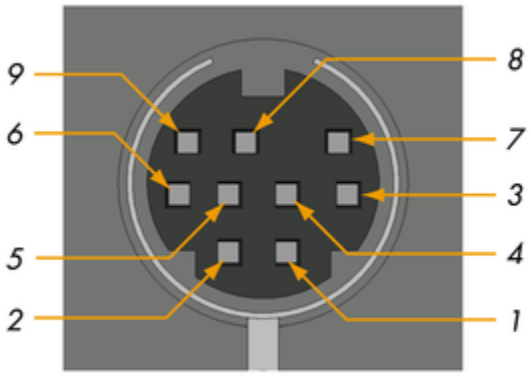


***Figure 33*** *: Beagle USB 480 Protocol Analyzer - Digital I/O Port Pinout*

**Table 5** : Digital I/O Cable Pin Assignments

| Pin Name | Color | Pin Number |
|---|---|---|
| Input 1 | Brown | Pin 1 |
| Input 2 | Red | Pin 2 |
| Input 3 | Orange | Pin 3 |
| Input 4 | Yellow | Pin 4 |
| Output 1 | Green | Pin 5 |
| Output 2 | Blue | Pin 6 |
| Output 3 | Purple | Pin 7 |
| Output 4 | Grey | Pin 8 |
| Ground | Black | Pin 9 |

The top of the Beagle USB 480 Protocol Analyzer has three LED indicators as shown in Figure 34.

- The green LED serves as an Analysis Port connection indicator. The green LED will be illuminated when the Beagle analyzer has been correctly connected to the analysis computer and is receiving power from USB.

- The amber LED serves as a Target Host connection indicator. The amber LED will be illuminated when the target host computer is connected to the analyzer.
- Finally, the red LED is an activity LED. Its blink rate is proportional to the amount of data being sent across the monitored bus. If no data is seen on the bus, but the capture is active, the activity LED will simply remain on.

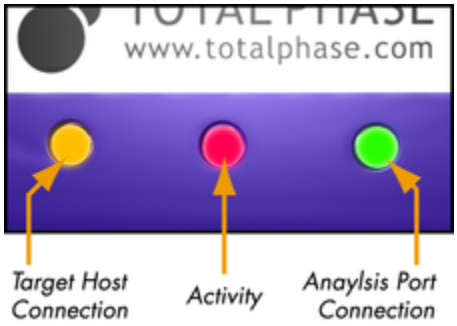


***Figure 34** : Beagle USB 480 Protocol Analyzer - LED Indicators*

Please check all the connections if the green or the amber LED fail to illuminate after the Beagle USB 480 analyzer has been connected to the analysis computer and the target host computer.

### 2.2.2 Digital I/O

Digital inputs allow users to synchronize external logic with the analyzed USB data stream. Whenever the state of an enabled digital input changes, an event will be sent to the analysis PC. The digital input may not oscillate at a rate faster than 30 MHz. Any faster and the events may not be passed to the PC. Also, when an active data packet is on the bus, only one input event will be recorded and sent back to the analysis PC. Once the packet has completed, the latest state of the lines (if changed) will be sent back to the PC. Digital inputs are rated for 3.3 V.

Digital outputs allow users to output events to external devices, such as an oscilloscope or logic analyzer, especially to trigger the oscilloscope to capture data. Digital outputs can be set to activate on various conditions that are described more thoroughly in Section 3.4.4. The digital outputs are rated to 3.3 V and 10 mA.

### 2.2.3 On-board Buffer

The Beagle USB 480 analyzer contains a 64 MB on-board buffer. This buffer serves two purposes. It helps buffer large data flows during real-time capture when the analysis computer can not stream the data off the Beagle analyzer fast enough. It is also used during a delayed-download capture to store all of the captured data.

### 2.2.4 Hardware Filters

The Beagle USB 480 analyzer provides six different hardware filters. These will filter out data-less transactions in the hardware, such as `IN` + `NAK` and `PING` + `NAK` combinations. The unwanted data is thrown away, reducing the amount of captured data on the device, the amount of analysis traffic back to the analysis PC, and the processing overhead on the analysis PC. A more detailed overview of the hardware filters is available in Section 3.4.5.

### 2.2.5 Current/Voltage Monitoring

For the Beagle USB 480 Power Protocol Analyzer, Standard and Ultimate Editions, the USB-A and USB-B capture inputs are rated 1A continuous current and 0 to 24V. The analyzer samples $V_{BUS}$ current/voltage measurements every 12 x $2^{16}$ clock cycles, which is approximately every 13.1 ms. The maximum current/voltage measurement error is ±50 mV and ±5 mA at 5 V and for current up to 500 mA. A more detailed overview of Current/Voltage Monitoring is available in Section 3.4.8.

**Disclaimer: When using the Beagle 480 USB Power Protocol Analyzer above the rated current and voltage, extreme caution is advised. Customers who choose to do so are at their own risk and may cause permanent damage to the analyzer. Total Phase is not liable for damages caused by applying current and voltage in excess of the warranted operating range.**

### 2.2.6 $V_{BUS}$ Trigger

The Beagle USB 480 Power Protocol Aanlzyer, Ultimate Edition, has the additional capability to trigger on a rise or drop in $V_{BUS}$ current or voltage. The USB-A and USB-B capture inputs of the analyzer are rated 1A continuous current and 0 to 24V. Although the analyzer can be configured to trigger on a current level from -3A to 3A, the continuous current should not exceed 1A. The voltage trigger level can be configured from 0 to 24V.

**Important: See the above disclaimer (Section 2.2.5) for operating the analyzer above the rated current and voltage.**

### 2.2.7 Signal Specifications / Power Consumption

#### Speed

The Beagle USB 480 Protocol Analyzer supports capture of all wired USB speeds. The analyzer has automatic speed detection as well as manual speed locking.

#### ESD Protection

The Beagle analyzer has built-in electrostatic discharge protection to prevent damage to the unit from high voltage static electricity.

#### Power consumption

When the Beagle analyzer is connected, it consumes a maximum of approximately 2.5 mA from the capture host. This is a minimal overhead in addition to the current draw of the target device. Note that if a capture target reports itself as a 100 mA device and draws almost all of that current, the Beagle analyzer's extra power consumption may cause the overall power consumption to be out of spec.

The Beagle analyzer consumes a maximum of approximately 180 mA.

## 2.3 Beagle USB 12 Protocol Analyzer

### 2.3.1 Connector Specification

On one side of the Beagle USB 12 monitor is a single USB-B receptacle. This is the **Analysis** side (Figure 35 ). This port connects to the analysis computer that is running the Beagle Data Center software.

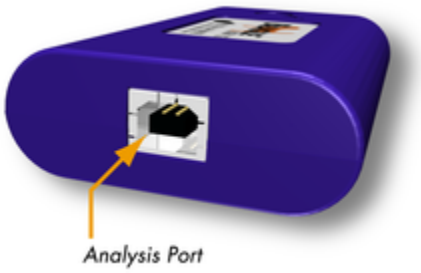


***Figure 35** : Beagle USB 12 Protocol Analyzer - Analysis Side*

On the opposite side is the **Capture** side (Figure 36), are a USB-A and USB-B receptacle. These are used to connect the target host computer to the target device. The target host computer can be the same computer as the analysis computer.

***Figure 36** : Beagle USB 12 Protocol Analyzer - Capture Side*

The **Capture** side acts as a USB pass-through. In order to remain within the USB 2.0 specifications, no more than 5 meters of USB cable should be used in total between the target host computer and the target device. The Beagle USB 12 monitor is galvanically isolated from the USB bus to ensure the signal integrity.

Please note, that on the **Capture** side, there is a small gap between the two receptacles. In this gap, two LED indicators are visible, a green one and an amber one, as shown in Figure 37. When the Beagle USB 12 monitor has been correctly connected to the analysis computer, the green LED will illuminate. When the Beagle USB 12 monitor is correctly connected to the target host computer, the amber LED will illuminate.

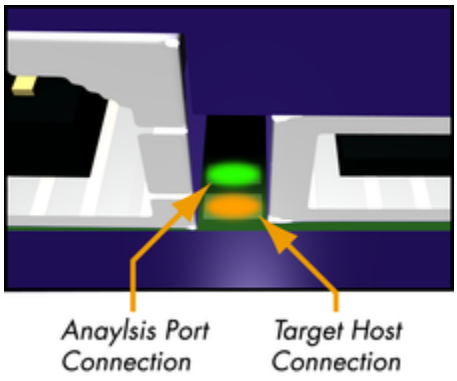


***Figure 37** : Beagle USB 12 Protocol Analyzer - LED Indicators*

Please check all the connections if the one or both LEDs fail to illuminate after the Beagle USB 12 monitor has been connected to the analysis computer or the target host computer.

### 2.3.2 Signal Specifications / Power Consumption

#### Speed

The Beagle USB 12 Protocol Analyzer supports full- and low-speed capture. It does not support high-speed protocols for capture. The downlink to the analysis PC must be high-speed.

#### ESD protection

The Beagle analyzer has built-in electrostatic discharge protection to prevent damage to the unit from high voltage static electricity.

#### Power consumption

The Beagle analyzer consumes a maximum of approximately 15 mA from the capture host. This is a minimal overhead in addition to the current draw of the target device. Note that if a capture target reports itself as a 100 mA device and draws almost all of that current, the Beagle analyzer's extra power consumption will cause the overall power consumption to be out of spec.

Furthermore, the Beagle analyzer consumes a maximum of approximately 125 mA of power from the analysis PC. However, it reports itself to the analysis PC as a low-power device. This reporting allows the Beagle analyzer to be used when its analysis port is connected to a bus-powered hub (which are only technically specified to supply 100 mA per port). Normally this extra amount of power consumption should not cause any serious problems since other ports on the hub are most likely not using their full 100 mA budget. If there are any concerns regarding the total amount of available current supply, it is advisable to plug the Beagle analyzer's directly into the analysis PC's USB host port or to use a self-powered hub.

## 2.4 Beagle $I^2C$/SPI/MDIO Protocol Analyzer

### 2.4.1 Connector Specification

The ribbon cable connector is a standard 0.100" (2.54 mm) pitch IDC type connector. This connector will mate with a standard keyed boxed header.

Alternatively, split cables are available which connects to the ribbon cable and provides individual leads for each pin with or without grabber clips.

### Orientation

The ribbon cable pin order follows the standard convention. The red line indicates the first position. When looking at your Beagle analyzer in the upright position (Figure 38 ), pin 1 is in the top left corner, and pin 10 is in the bottom right corner.

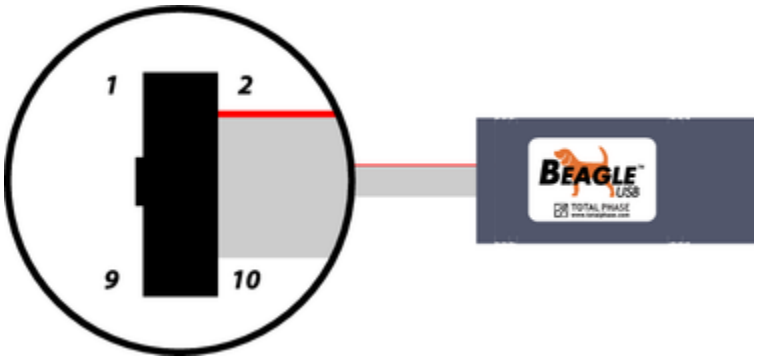


***Figure 38*** *: The Beagle I²C/SPI/MDIO Protocol Analyzer in the upright position.*
*Pin 1 is located in the upper left corner of the connector, and Pin 10 is located in the lower right corner of the connector.*

If you flip your Beagle analyzer over (Figure 39 ) such that the text on the serial number label is in the proper upright position, the pin order is as shown in the following diagram.

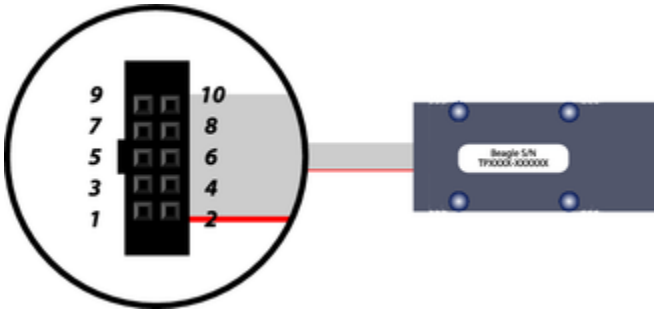

***Figure 39*** *: The Beagle I²C/SPI/MDIO Protocol Analyzer in the upside down position.*
*Pin 1 is located in the lower left corner of the connector and Pin 10 is located in the upper right corner of the connector.*

### Order of Leads

1. SCL
2. GND
3. SDA
4. NC/+5V
5. MISO
6. NC/+5V
7. SCLK/MDC
8. MOSI/MDIO
9. SS
10. GND

### Ground

**GND (Pin 2):**
**GND (Pin 10):**

It is imperative that the Beagle analyzer's ground lead is connected to the ground of the target system. Without a common ground between the two, the signaling will be unpredictable and communication will likely be corrupted. Two ground pins are provided to ensure a secure ground path.

### I²C Pins

**SCL (Pin 1):**

Serial Clock line – the signal used to synchronize communication between the master and the slave.

**SDA (Pin 3):**

Serial Data line – the bidirectional signal used to transfer data between the transmitter and the receiver.

### SPI Pins

#### SCLK (Pin 7):

Serial Clock – control line that is driven by the master and regulates the flow of the data bits.

#### MOSI (Pin 8):

Master Out Slave In – this data line supplies output data from the master which is shifted into the slave.

#### MISO (Pin 5):

Master In Slave Out – this data line supplies the output data from the slave to the input of the master.

#### SS (Pin 9):

Slave Select – control line that allows slaves to be turned on and off via hardware control.

### MDIO Pins

#### MDC (Pin 7):

Management Data Clock – control line that is driven by the STA and synchronizes the flow of the data on the MDIO line.

#### MDIO (Pin 8):

Management Data Input/Output – the bidirectional signal used to transfer data between the STA and the MMD.

### Powering Downstream Devices

It is possible to power a downstream target, such as an $I^2C$ or SPI EEPROM with the Beagle analyzer's power (which is provided by the analysis PC's USB port). It is ideal if the downstream device does not consume more than 20-30 mA. The Beagle analyzer is compatible with USB hubs as well as USB host controllers. Bus-powered USB hubs are technically only rated to provide 100 mA per USB device. If the Beagle analyzer is directly plugged into a USB host controller or a self-powered USB hub, it can theoretically draw up to 500 mA total, leaving approximately 375 mA for any downstream

target. However, the Beagle analyzer always reports itself to the host as a low-power device. Therefore, drawing large amounts of current from the host is not advisable.

### 2.4.2 Signal Specifications / Power Consumption

#### Speed

The Beagle $I^2C$/SPI/MDIO is capable of monitoring $I^2C$ bus bit rates of up to 4 MHz, SPI bit rates of up to 24 MHz, and MDIO bit rates of up to 2.5 MHz. Both $I^2C$ and MDIO monitoring can sustain their respective maximum speeds, however SPI monitoring at the maximum bit rate may not be possible for sustained traffic. The exact limitations of SPI monitoring are dependent on the target bus conditions and the CPU of the host PC. For example, the worst-case situation is a sustained sequence of short SPI packets at the maximum bus bit rate of 24 MHz.

It is important to note that in order to properly capture $I^2C$, SPI, or MDIO signals, the sampling rate must be set properly. For SPI or MDIO monitoring, the minimum requirement for the sampling rate is twice the bus bit rate. For $I^2C$ monitoring, the sampling rate should be 5-10 times the bus bit rate. For further details on this refer to Section 3.4.6.3.

#### Logic High Levels

All signal levels should be nominally 3.3 V (+/- 10%) logic high. This allows the Beagle analyzer to be used with both TTL (5 V) and CMOS logic level (3.3 V) devices. A logic high of 3.3 V will be adequate for TTL-compliant devices since such devices are ordinarily specified to accept logic high inputs above approximately 3 V.

#### ESD protection

The Beagle analyzer has built-in electrostatic discharge protection to prevent damage to the unit from high voltage static electricity. This adds a small amount of parasitic capacitance (approximately 15 pF) to the signal path under analysis.

#### Power Consumption

The Beagle analyzer consumes approximately 125 mA of power from the analysis PC. However, it reports itself to the analysis PC as a low-power device. This reporting allows the Beagle analyzer to be used when its analysis port is connected to a bus-powered hub (which are only technically specified to supply 100 mA per port). Normally this extra amount of power consumption should not cause any serious problems since other ports on the hub are most likely not using their full 100 mA budget. If there are any concerns regarding the total amount of available current supply, it is advisable to plug the Beagle analyzers directly into the analysis PC's USB host port or to use a self-powered hub.

## 2.5 USB 2.0

All Beagle analyzers are high-speed USB 2.0 devices. They require a high-speed USB 2.0 host controller for the analysis data connection.

## 2.6 Temperature Specifications

The Beagle analyzers are designed to be operated at room temperature (10-35°C). The electronic components are rated for standard commercial specifications (0-70°C). However, the plastic housing, along with the ribbon and USB cables, may not withstand the higher end of this range. Any use of the Beagle analyzer outside the room temperature specification will void the hardware warranty.

# 3 Device Operation

## 3.1 Electrical Connections

### 3.1.1 Beagle USB Protocol Analyzers

The Beagle USB analyzer's analysis port must be connected to the analysis computer through a USB cable. The Capture side of the Beagle analyzer must be placed on the USB to be monitored. Normally, this is accomplished by placing the Beagle analyzer in-line between the USB device and host being monitored. In other words, the bus to be monitored goes through the Beagle USB analyzer. To properly accomplish this connection, connect the target host to the USB-B receptacle on the Capture side of the Beagle USB analyzer, and connect the target device to the USB-A receptacle on the Capture side of the Beagle USB analyzer. See Section 2.2.1 for more details. This is the setup illustrated in panels a-c of Figure 40.

In some cases, the target bus is fully internal to an embedded system. If so, it is simply necessary to tap off the lines through the use of a parallel connector. One can plug in the tapped off cable into either the Target host or Target device port of the analyzer; both are equivalent. This is illustrated in Figure 40 d.

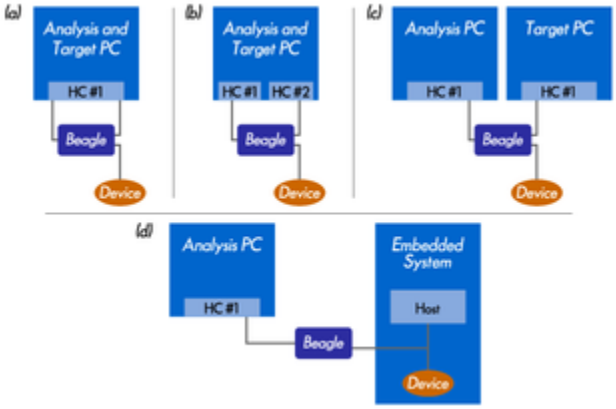


***Figure 40** : Beagle USB Protocol Analyzer Connections Beagle USB analyzer may be connected to the same bus as it is monitoring (panel a), or to a different bus (panel b). Multiple host controllers may reside in separate host controllers (panel c). Panel d shows the case of sniffing a self-contained embedded bus.*

The connections of the Beagle USB analyzer are complicated somewhat by the fact that the Beagle analyzer is monitoring USB signals and then communicating the monitored data back though another USB port. Thus, the issue of the host broadcasting, as

described in Section 1.1.2, comes into play. Because all Beagle analyzers use high-speed USB communication, this issue is only pertinent when using one of the high-speed capable protocol analyzers (such as the Beagle USB 480 Protocol Analyzer or the Beagle USB 5000 SuperSpeed Protocol Analyzers) to monitor a high-speed device. If the Beagle analyzer's analysis port is connected to the same host controller as a high-speed device that it is monitoring (Figure 40 a) then the Beagle analyzer will end up sniffing some of its own traffic. This is especially true if the Beagle analyzer is configured to stream back bus traffic to the PC in real time! This will be seen in the capture as many `IN` packets to the Beagle analyzer's device address with occasional downstream handshake packets.

This phenomenon has two negative consequences. The extra traffic on the capture bus from the Beagle analyzer may make it difficult to locate the USB traffic of interest within the volume of data captured. Additionally, the bus traffic for Beagle analyzer will reduce the bandwidth available to other USB devices on the bus.

There are a number of ways to deal with this issue.

- One method for dealing with this problem is install another USB host controller to the computer and connect one host controller to the analysis port of the Beagle analyzer and use the other host controller to communicate with the host and device under test (Figure 40 b). Downstream USB packets are only broadcast on USB links on the same host controller, so this technique is another way to ensure that the Beagle analyzer's traffic is not seen on the capture side of the analyzer. The disadvantage is that the PC must spend processing time for communicating both with the target device as well as the Beagle analyzer.
- The preferred method is to connect separate computers to the analysis port and to the target host port of the Beagle analyzer (Figure 40 c). This puts the analysis end of the Beagle analyzer on a different bus, ensuring that its traffic is not seen on the capture side of the analyzer. Furthermore, the analysis PC can have full resources to process the incoming data, and the test PC will not be encumbered by the analysis software.

Note: All of the USB ports on most computers are on a single host controller, so connecting to a different USB port is not sufficient. Installing a PCI, PCI Express, or PC Card USB controller card will ensure there is a second USB host controller on the computer.

If the user is constrained to the scenario illustrated in Figure 40a, there are two features of the Beagle USB 480 and Beagle USB 5000 analyzer that help mitigate the dilemmas previously outlined.

- One feature is a hardware filtering option that runs on the Beagle analyzer to filter packets directed to the Beagle analyzer's device address. These packets will be filtered out from the capture by the hardware, so it will not be sent back through the analysis port. This option does not entirely remove the Beagle analyzer's

traffic from the monitored bus, but it will definitely minimize the analyzer's effect on the bus since the `IN` and `ACK` tokens sent to the analyzer will not again appear in the analysis traffic. In situations where the maximum bandwidth is required by the target device, avoid using this option.

- The second feature is the ability to perform a delayed-download capture. In this capture mode, the capture data is not streamed out of the analysis port of the Beagle analyzer until after the analyzer has stopped monitoring the bus. This greatly reduces the amount of USB traffic going to the Beagle analyzer while the capture is active. These features are mentioned later in this section where appropriate.

### 3.1.2 Beagle I²C/SPI/MDIO Protocol Analyzer

The Beagle I²C/SPI/MDIO analyzer uses a standard USB cable to connect the protocol analyzer to the analysis computer. The data line(s), clock, and ground of the communication protocol in question must be properly connected to the Beagle analyzer's data line(s), clock, and ground, respectively.

## 3.2 Software Operational Overview

There are a series of steps required for a successful capture. These steps are handled by the Beagle Data Center software automatically, but must be explicitly followed by an application programmer wishing to write custom software. The application programmer interface (API) is documented extensively in Section 6, but the following is meant to provide a high-level overview of the operation of the Beagle analyzers.

1. Determine the port number of the Beagle analyzer. The function `bg_find_devices()` returns a list of port numbers for all Beagle analyzers that are attached to the analysis computer.
2. Obtain a Beagle handle by calling `bg_open()` on the appropriate port number. All other software operations are based on this Beagle handle.
3. Configure the Beagle analyzer as necessary. The API documentation provides complete details about the different configurations.
4. Start the capture by calling the `bg_enable()` function.
5. Retrieve monitored data by using the read functions that are appropriate for the monitored bus type. There are different functions available for retrieving additional data such as byte- and bit-level timing.
6. End the capture by calling the `bg_disable()` function. At this point the capture is stopped, and no new data can be obtained.
7. Close the Beagle handle with the `bg_close()` function.

If the Beagle analyzer is disabled and then re-enabled it does not need to be re-configured. However, upon closing the handle, all configuration settings will be lost.

Example code is available for download from the Total Phase website. These examples demonstrate how to perform the steps outline above for each of the serial bus protocols supported.

## 3.3 Beagle USB 5000 Protocol Analyzer Specifics

### 3.3.1 Heat Dissipation

Power is provided to the Beagle USB 5000 Protocol Analyzer by an external AC adapter. During its operation, it is normal for the Beagle analyzer to become warm to the touch. To help dissipate heat, the analyzer includes an internal fan which automatically turns on as needed.

### 3.3.2 Receiver Termination Detection

Termination resistors on the SuperSpeed receivers are required for good signal integrity. The SuperSpeed transmitter uses the presence of the termination resistors to detect the presence of a SuperSpeed receiver. According to the USB 3.0 specification, if the termination resistor is not detected, the SuperSpeed transmitter should not send any SuperSpeed packets.

The Beagle USB 5000 analyzer uses a sophisticated algorithm to detect the presence of the termination resistor. When the termination resistor is detected, the Beagle analyzer will illuminate the RxTerm activity LED on the front panel and apply a termination resistor on the lines presented to the transmitter.

Receiver detection can take up to 500 us, however this should have no affect on the USB 3.0 data. According to the USB 3.0 specification, the transmitter should not send any USB 3.0 data until the termination resistor is detected. The net effect of the delay introduced by the detection system would be as if the cable insertion was delayed by 500 us and should have no affect on the USB 3.0 data.

The Beagle USB 5000 analyzer's receiver termination detection system is set to auto-detect by default. It is also possible to manually set the receiver termination to be always on or always off in the upstream or downstream direction. If the termination resistor is set manually, the auto-detection system will be turned off. Once the handle to the Beagle analyzer is closed, the state of the receiver termination detection system will revert to the default setting of auto-detect.

If the termination resistors are forced on, the Beagle analyzer will apply termination resistors to the lines it presents to the transmitter, regardless of the true state of the target receiver termination. The RxTerm LED will be on, to indicate that the lines to the transmitter have been terminated.

If the termination resistors are forced off, the Beagle analyzer will not apply a termination resistor, regardless of whether the receiver has terminated the lines. The RxTerm LED will be off, to indicate that the lines to the transmitter have not been terminated.

While the analyzer is in auto-detection mode and is in the process of testing for the presence or absence of a receiver, it is not able to pass USB 3.0 data to the target receiver. Conversely, while data is being sent to the target receiver, it is not possible for the analyzer to detect the presence or absence of the termination resistor.

**Note:**If a device is removed while data is being sent, the Beagle analyzer will prioritize capturing the data and will not be able to detect the absence of the termination immediately. In this situation, the Beagle analyzer will be presenting receiver termination on its link, which does not accurately reflect the state of the link with the receiver. The transmitter should quickly detect that there is a problem with the link and stop transmitting data. At this time, the Beagle analyzer will be able to detect the absence of the termination resistor and remove its termination accordingly. When the termination is removed, the RxTerm LED will be turned off.

The receiver termination detection system is always operating as long as the Beagle analyzer is powered and in auto-detect mode, regardless of whether a capture is active. As long as the analyzer is set to auto-detect, the analyzer will properly reflect the termination of the receivers on the bus. This allows SuperSpeed hosts and devices to properly transition through the LTSSM, even outside the scope of an active capture.

### 3.3.3 Polarity Detection

The polarity of a SuperSpeed differential pair is mutable to provide flexibility in the circuit layout design. As a consequence of this the receiver must correctly detect the polarity of the transmitters during the link training in order to properly interpret the subsequently transmitted data.

The Beagle USB 5000 Protocol Analyzer has a robust, auto-detection system to correctly identify the polarity of the target system. By default, this auto-detection system is enabled. As an added feature, the polarity of each SuperSpeed differential pair can be manually configured as well.

When the polarity is manually controlled, the analyzer will force the desired setting in the hardware. At this point, the data received will be of the forced polarity and cannot be changed in the software. Once the handle to the Beagle analyzer is closed, the state of the polarity detection system will revert to the default setting of auto-detect.

Even when the polarity is manually controlled, the analyzer will still track what it believes to be the correct polarity. Reverting the polarity setting to auto-detect will cause the polarity on the analyzer to be set to this auto-detected configuration.

Furthermore, the polarity detection system is running even when a capture is not active. This ensures that the analyzer is always up-to-date with the current requirements of the devices, even when these settings change outside of the scope of an active capture.

It is important to note that the Beagle analyzer is detecting the polarity on the tapped signal and does not in any way influence or modify the signal sent between the transmitter and receiver. By manually changing the polarity of the lines, the analyzer is simply changing its perception of the tapped signal and is not modifying the data sent between the transmitter and receiver.

### 3.3.4 Data Scrambling Detection

Data scrambling is used to minimize interference on the data lines. When data scrambling is enabled, nearly all symbols from the transmitter are scrambled before they are 8b/10b encoded. The receiver must then decode the 8b/10b and descramble the data appropriately to properly interpret the data.

Data scrambling is implemented by applying an XOR to the transmitted data with a pseudo-random number. Receivers must then apply an XOR with the same number to the received data in order to descramble the data. This pseudo-random number is generated through the use of an linear feedback shift register (LFSR) that is updated upon the transmission/reception of every symbol on the bus (with the exception of COM and SKP symbols).

It is very important that the transmitter and receiver's LFSR stay synchronized, otherwise data will be incorrectly descrambled. To facilitate the synchronization of the LFSRs, COM symbols will reset the LFSR on the transmitter and receiver. Each training sequence packet (TSEQ, TS1, TS2) starts with at least one COM symbol, and will therefore reset and synchronize the transmitter and receiver scrambling state.

According to the USB 3.0 specification, the transmitter informs the receiver of its scrambling mode through a single bit in the TS1 and TS2 packets. In order for a receiver to interpret the transmitted data correctly, it must properly decode this bit and enable/disable its descrambler as appropriate.

The Beagle USB 5000 Protocol Analyzer is able to reliably detect the state of data scrambling and the reset events. Using its own LFSR, the Beagle analyzer can robustly decode and descramble the tapped data.

By default data scrambling detection will be set to auto-detect. It is also possible to manually turn data scrambling on or off for each of the analyzed streams.

The manual control of the scrambler can be especially useful in situations of bad signal integrity. When poor signals are on the bus, it is possible for the scrambling control bit of the TS1 or TS2 to be corrupted, and thus unintentionally mis-inform the receiver of the true scrambling mode. In the worst case, this corruption occurs on the final TS2 transmitted, and puts the analyzer in a bad state for all subsequent data. By forcing the scrambling mode, users can test and correct for this rare, but possible, error.

When the data scrambling is manually controlled, the analyzer will force the desired setting in the hardware. At this point, the data received will be descrambled according to the forced setting and cannot be changed in the software. Once the handle to the Beagle

analyzer is closed, the state of the scrambling system will revert to the default setting of auto-detect.

Even when the scrambling is manually controlled, the analyzer will still track what it believes to be the proper scrambling mode, as well as the proper state of the LFSR. Reverting the scrambling setting to auto-detect will cause the scrambling on the analyzer to be set to this auto-detected configuration.

Furthermore, the data scrambling system is running even when a capture is not active. This ensures that the analyzer is always up-to-date with the current requirements of the devices, even when these settings change outside of the scope of an active capture.

Please note that according to the USB 3.0 specifications, training sequences are never scrambled. Consequently, even if the data scrambling is manually turned on, training sequences will not be scrambled. Similarly, K-symbols are never scrambled, even if the user manually turns the scrambling on.

It is important to note that the Beagle analyzer is detecting the data scrambling on the tapped signal and does not in any way influence or modify the signal sent between the transmitter and receiver. By manually changing the data scrambling of the data, the analyzer is simply changing the perception of the tapped signal and not modifying the data sent between the transmitter and receiver.

### 3.3.5 Digital Inputs

For the Beagle USB 5000 analyzer, the digital inputs provide a means for users to trigger the capture or insert events into the data stream.

**WARNING: The USB 3.0 Input is only rated for 1.8 V.** The USB 3.0 input of the Beagle USB 5000 analyzer have been optimized for maximum edge performance at 125 MHz. Consequently, it cannot tolerate voltage signals higher than 1.8 V. **Applying signals with higher voltage will damage your analyzer and is not covered by the warranty.**

For USB 3.0, there is a single external input that can be enabled by configuring matching in the device settings. The USB 3.0 input can be used to trigger the capture and insert events into the data stream. After the capture has been triggered, subsequent external input events can be inserted into the capture.

The USB 3.0 input can capture external signals up to 125 MHz (8 ns pulse width). However, in the interest of preserving capture fidelity, the analyzer may throttle the external input signal if the rate of the input events are judged to be too high.

**WARNING: The USB 2.0 Input is only rated for 3.3 V. Applying signals with higher voltage will damage your analyzer and is not covered by the warranty.**

The USB 2.0 external inputs are available on pins 1 through 4 on the Mini-DIN 9 connector. These digital input lines are 3.3 V tolerant. Each input can be configured individually to be monitored in the capture. Each input can also be independently

configured to trigger the capture on either the rising edge, falling edge, or both. A USB 2.0 digital input does not need to be monitored to function as a trigger.

Besides the ability to trigger the capture and the voltage level, the USB 2.0 inputs have the same basic behavior as the digital inputs of the Beagle USB 480 analyzer described in Section 3.4.3.

### 3.3.6 Digital Output

The digital outputs provide a mechanism to synchronize the Beagle USB 5000 analyzer with other devices, such as an oscilloscope or logic analyzer, on events of interest.

**WARNING: The USB 3.0 Output is only rated for 1.8 V.** The USB 3.0 output of the Beagle USB 5000 analyzer have been optimized for maximum edge performance at 125 MHz. Consequently, it cannot tolerate voltage signals higher than 1.8 V. **Applying signals with higher voltage will damage your analyzer and is not covered by the warranty.**

The USB 3.0 external output has a short latency of 50 to 75 ns from when a trigger occurs and when the output is asserted on the SMA connector. The output can be asserted by a sophisticated matching system which provides a spectrum of functionality, from simple matches all the way to complicated multi-state triggers.

The behavior of the output is configurable and can be set to:

- Set Low
- Set High
- Positive Pulse
- Negative Pulse
- Toggle (Initially Low)
- Toggle (Initially High)

When configured as a positive or negative pulse, the pulse width is 24 ns. For FW versions before v1.10, the pulse width is 40 ns.

**WARNING: The USB 2.0 Output is only rated for 3.3 V. Applying signals with higher voltage will damage your analyzer and is not covered by the warranty.**

The USB 2.0 external outputs are available on pins 5 through 8 on the Mini-DIN 9 connector. These outputs behave similarly to the digital outputs of the Beagle USB 480 analyzer as described in Section 3.4.4, except that the output level is 3 V.

### 3.3.7 Cross-Analyzer Sync

#### Overview

Multiple point analysis of a USB system is extremely helpful in a variety of development scenarios. The most obvious example is hub development, where traffic is sent between the hub and the host, as well as between the hub and the device.

Normally, clock drift prevents the reliable correlation of captured data from two different analyzers. Synchronizing capture events (start, trigger, stop) on multiple analyzers also proves difficult.

Beagle Cross-Analyzer Sync solves the challenges posed by multiple point analysis, allowing users to easily and reliably monitor both sides of a USB hub, or any number of points in a USB system. Two HDMI ports on the back of the Beagle USB 5000 analyzer (Section 2.1.2.3 ) allow two or more analyzers to synchronize their capture timestamps, as well as their capture start, capture trigger, and capture stop events.

#### Setup

Cross-Analyzer Sync is easy to use. Connect the SYNC OUT port on one Beagle USB 5000 analyzer to the SYNC IN port on another analyzer. Once two analyzers are connected, adding more analyzers to the chain is simple. The following steps describe the setup process.

1. Connect SYNC OUT on one analyzer to SYNC IN on another analyzer to begin a sync chain.
2. Connect SYNC OUT on the last analyzer in the chain to SYNC IN on an unconnected analyzer to extend the sync chain. Repeat this step as desired to expand the sync chain.

#### Start Capture

Once a Cross-Analyzer Sync chain is setup, follow the steps below to start a synchronized capture on each of the analyzers:

1. Follow steps 1-4 in Section 3.2 to place an analyzer in the SYNC_STANDBY state. The order of analyzers on which this step is performed is not important.

2. Repeat step 1 for each analyzer in the sync chain. Capture will automatically start on each analyzer once step 1 is completed for every analyzer in the sync chain.

### Trigger Capture

Cross-Analyzer Sync allows multiple analyzers to be triggered, advancing from pre-trigger to post-trigger, in a synchronized manner. An analyzer connected via Cross-Analyzer Sync will output a trigger signal to other analyzers in the chain when its capture is triggered.

The trigger signal can be ignored on a per-analyzer basis by software configuration (Section 6.8.6.1 ), allowing an analyzer to remain in pre-trigger even if other analyzers connected by Cross-Analyzer Sync have triggered. An analyzer configured to ignore incoming trigger signals will still output a trigger signal to other analyzers on its own capture trigger.

### Stop Capture

Cross-Analyzer Sync allows multiple analyzers to stop capturing at the same time. An analyzer connected via Cross-Analyzer Sync will output a stop signal to other analyzers in the chain when its capture is stopped.

The stop signal can be ignored on a per-analyzer basis by software configuration (Section 6.8.6.1 ), allowing an analyzer to continue capture even if other analyzers connected by Cross-Analyzer Sync have stopped capture. An analyzer configured to ignore incoming stop signals will still output a stop signal to other analyzers on its own capture stop.

### Software Release

An analyzer can be completely released from Cross-Analyzer Sync by software configuration before capture (Section 6.8.6.1) or during capture (Section 6.8.6.2). An analyzer released from Cross-Analyzer Sync by software will:

1. Not wait on other analyzers connected by Cross-Analyzer Sync when capture is started.
2. Ignore Cross-Analyzer stop and Cross-Analyzer trigger signal inputs.

3. Not output Cross-Analyzer stop or Cross-Analyzer trigger signals.

**Notes**

Do not change the sync configuration by attaching or detaching HDMI cables from an analyzer when capture is in progress. Disrupting HDMI cables during capture may result in termination of capture and disconnection of an analyzer from software.

Cross-Analyzer Sync HDMI ports are only available on hardware v2.00 or later.

### 3.3.8 Match/Action System

The Beagle USB 5000 Protocol Analyzer features a multi-tiered matching system that can perform one or more actions in response to a match. The USB 3.0 matching system is separate from the USB 2.0 matching system.

**USB 3.0 Matching**

Within the USB 3.0 Matching framework, there are multiple tiers of matching. The first level is Simple Matching which can match the occurrence of general packet types, events, or errors and trigger the capture or assert the external output in response.

The next level is Complex Matching, which provides the a state-based facility to match specific packet types and data patterns in addition to specific events. The standard Beagle USB 5000 analyzer provides a single state and limited matching facilities. The Advanced trigger option, extends the Complex Matching framework with multiple states and extended matching facilities to build complex state machines.

**USB 3.0 Simple Matching**

With USB 3.0 simple matching, the Beagle USB 5000 analyzer is capable of matching:

- Link Commands
- Header Packets
- Data Payloads
- CRC Errors (CRC32, CRC12, CRC5 LCW, CRC5)
- Training Sequences (TS1, TS2, TSEQ)
- $V_{BUS}$ Detected
- External Input (rising, falling)
- Reverse Polarity

- Termination Detected
- Scrambling Disabled
- LPFS
- PHY Error

PHY Error is a special-case bus event that will match the following errors:

- Disparity Error
- Elastic Buffer under-run or over-run
- 8b/10b Decode Error

While the PHY Errors collapses these 4 errors into a single match, it is possible to distinguish some of the different errors in the captured data. When an elastic buffer under-run error occurs, an EDB symbol (K28.3) is inserted into the data stream to fill the under-run. When an 8b/10b Decode Error occurs, a SUB symbol (K28.4) is substituted in place of the bad 10b symbol in the data stream.

It is possible to select multiple events to match the simple trigger. However, since a capture can only be triggered once, in the case of multiple selected events, the first of any of the selected events will trigger the capture.

When a match occurs in the Simple Triggers, it is also possible to assert the External Output. The output can be asserted only once, when the trigger occurs, or every time the simple triggers match.

**USB 3.0 Complex Matching**

The USB 3.0 complex matching system provides additional matching capabilities. There are two variants of the complex matching system. The basic variants included with the Standard Beagle USB 5000 analyzer offers a single state with one upstream data match unit, one downstream data match unit, and one event match unit.

An optional advanced matching package extends the functionality of the complex matching by providing up to eight states with up to three upstream data match units, three downstream data match units, one event match unit, and one timer match unit per state. Actual number of match units available per state will depend on the remaining resources available.

**States**

The Complex Matching system provides up to 8 states. Each state is comprised of one or more match units. Each match unit defines a specific matching criteria. Since a state can transition to multiple other states, the order of precedence is significant. If multiple

match units match at the same time, the first match unit will have priority as to which state is transitioned to.

### Data Match Unit

A data match unit can match specific data packet types or data in either the upstream direction or downstream direction. The types of data that can be matched are: link commands, header packets, data packets, qualified data packets, and training sequences. Within each of these packet types, specific subtypes or data patterns can be defined. For example, under link commands, only "LGOOD_3" can be specified. For any of the packet types, valid or invalid CRCs can be specified as part of match criteria of the match unit.

**Table 6** : Match Supported Packets

| Packet Type | Packet |
|---|---|
| Link Command | LGOOD_n<br>LGOOD_[0-7]<br>LRTY<br>LBAD<br>LCRD_x<br>LCRD_[A-D]<br>LGO_Ux<br>LGO_U[1-3]<br>LAU<br>LXU<br>LPMA<br>LUP<br>LDN |
| Header Packet | Link Management Packet<br>Transaction Packet<br>Data Packet Header<br>Isochronous Timestamp Packet |
| Data Packet | Data Packet |
| Training Sequence | TSEQ<br>TS1<br>TS2 |

Part of the criteria of a match unit is the ability to match for the negative criteria. For example, a match unit can be set to match **LGOOD_n** only or match any other Link Command packet **NOT LGOOD_n**.

The negative criteria normally only applies within a group of packets. In the previous example, matching **NOT LGOOD_n** will only match any other link command that is not an **LGOOD_n**. However, sometimes it is useful to be able to match any packet of any type that is not an **LGOOD_n**. To do this, the match unit has the ability to match or not

match any other packet types. For example, it is possible to match a situation where an ITP is followed by any packet that is not an **LGOOD_n** by setting the match unit to match any packet that is **NOT LGOOD_n**.

For data packets, a specific data pattern is the basis of the match. It is possible to restrict the scope of the match by limiting the match criteria to a specific device, stream, endpoint, and/or the length of the data packet.

#### Event Match Unit

The event match unit can match any of the following events in either the upstream or downstream direction: LFPS, Polarity Inversion, RX Termination, and Disable Scrambling. The event match unit can also match $V_{BUS}$ detection and the external input.

#### Timer Match Units

The timer match unit will match after a specific amount of time has elapsed. The timer begins upon entering the state which declared it, and can run anywhere from a few nanoseconds to half a minute. Using a multi-state trigger, a match can occur a set amount of time after a specific event.

#### Counters

The execution of an a match units action can also be controlled by a counter. All match units can be configured to execute its action only after a specific number of matches have been made. For example, a match unit may not trigger the capture until 20 TS1 packets are seen.

A match unit can be further configured to trigger on every match after a set number of matches. For example, a match unit can be configured to filter out all TSEQ after the first 50. In this way, only the first 50 TSEQ packets would be seen in the capture, and all subsequent TSEQs would be filtered out.

#### Actions

When a match units successfully makes a match, it can perform one or more actions. The available actions are triggering the capture, asserting the external output, filtering the matching data out, or going to a different state. The ability to go to a different state is only available in the Advanced Complex Triggers option.

Each individual Match Unit in a state can have different actions which provides flexibility of defining a complex multi-state matching trigger.

### USB 2.0 Matching

The USB 2.0 matching system provides the user the capability of matching events, packet types, packet data, or external input. The functionality of this system is based on the Beagle USB 480 Protocol Analyzer's system with the addition of the ability for the inputs and outputs to trigger the capture.

### 3.3.9 Capture Settings

The Beagle USB 5000 analyzer is the only USB protocol analyzer with the ability to capture USB 3.0 and USB 2.0 data and stream it in real-time to the Analysis PC for interactive display.

#### Real-time USB 3.0 Capturing

It is important to note that the downlink from the analyzer to the Analysis PC is only high-speed USB operating at 480 Mbps. Theoretically, the SuperSpeed USB front-end is able to operate at up to 5 Gbps. Given this disparity, there are several key technologies involved in providing as close to a real-time experience as possible.

When capturing data, the Beagle analyzer has a highly efficient data stream that adds very little overhead to the USB 3.0 and 2.0 data streams it is monitoring. This low overhead reduces the amount of data that needs to be uploaded.

The high-speed USB driver has been highly optimized to deliver as close to the theoretical maximum performance as possible. The Beagle analyzer is able to sustain download speeds of up to 40 MB/s. Given that USB traffic being monitored tends to occur in bursts, the Beagle analyzer is able to catch up quickly, and eventually be up-to-date, during gaps in the USB bus being monitored.

When there are large streams of the USB 3.0 data, the Beagle USB 5000 analyzer will make full use of its 2 GB buffer to keep pace with the bus under analysis. However, the analyzer may reach a point where the memory buffer will be filled and no additional data can be captured. At this point the capture of new data will stopped, and all remaining data will be downloaded to the PC.

#### Capture Modes

The Beagle USB 5000 analyzer is able to capture USB 3.0 or USB 2.0 data. With an additional upgrade, the Beagle analyzer can capture both USB 3.0 and USB 2.0 simultaneously.

#### Capture Buffer

The Beagle USB 5000 analyzer includes a standard 2 GB memory buffer. In the future, the amount of memory can be upgraded to 4 GB.

Not all the memory needs to be used when capturing USB data. The amount of memory allocated to capturing data in the Beagle analyzer can be configured to limit the size of the overall capture.

The memory dedicated to capturing data can be further allocated into pre-trigger and post-trigger buffers.

#### Infinite Capture

Since the Beagle analyzer constantly streams data to the Analysis PC, it is possible for the Beagle analyzer to be configured for infinite capture. As long as the Beagle analyzer is able to keep up with the USB 3.0 traffic, it can theoretically continue capturing data indefinitely. The total capture size will eventually be limited by the amount of memory in the Analysis computer.

### 3.3.10 Capture Issues

#### Signal Integrity

The Beagle USB 5000 analyzer achieves bit lock and symbol lock with the signal received from the host and device like any other device. Bad signals on the USB 3.0 bus can cause problems for the analyzer. For this reason, it is recommended that high-quality, short cables are used with large gauge conductors.

## 3.4 Beagle USB 480 Protocol Analyzer Specifics

Aside from standard real-time capture, the Beagle USB 480 analyzer provides a number of other features. These features include bus event monitoring, digital inputs and outputs, hardware filtering, as well as multiple capture modes.

### 3.4.1 Bus Events

The Beagle USB 480 analyzer provides users with insight into events that occur on the bus. These bus events include suspend, resume, reset, speed changes (including high-speed handshake), and connect/disconnect events. Furthermore, events that are unexpected (i.e., do not conform to the USB spec) are tagged with a specific status code to bring that to the attention of the user. The Beagle USB 480 analyzer also has the ability to identify imperfect resets, like a Tiny J associated with the high-speed handshake. A Tiny J (or K) may also be tagged when not in a high-speed handshake situation if the reset is not fully at an SE0, but is instead floating above the high-speed receiver threshold. This allows users to see if the host is driving a reset signal that is close enough to ground voltage. Alternatively, if this amount of detail on reset signals is not desired, the auto speed-detection could be disabled, and locked to the specific speed of interest.

Where applicable, bus events are also returned with a duration. In most cases, this duration is self-explanatory, such as in the duration of a Chirp K or a keep-alive bus state. However, some clarification is required for the reported duration of suspend and resume events.

For suspend events, the Beagle USB 480 analyzer will return the duration of the event as it is measured from the devices perspective. For example, in a case where the bus is idle for 8 ms, the analyzer will return 5 ms for the duration of the suspend; this

corresponds with the fact that the device can only enter the suspend state after 3 ms of bus idle and is therefore suspended for 5 ms.

For resume events, the Beagle analyzer will return the duration of the K portion of the resume signaling. Rather than combine this duration with the ensuing SE0, this scheme provides users with the ability to determine the individual durations of each segment of the resume event. Specifically the user can refer to the start time of the resume event, the duration of the resume event (time of K state), and the start time of the subsequent event or packet.

For more details on USB bus events refer to Section 1.1.2 and the USB 2.0 spec.

### 3.4.2 OTG Events

The Beagle USB 480 analyzer has the ability to detect On-The-Go (OTG) events. These events include the Host Negotiation Protocol (HNP) and each stage of the Session Request Protocol (SRP). For more details on these protocols, see Section 1.1.3.11.

A HNP event will be returned upon seeing the correct initial conditions, and then detecting a correctly timed SE0 followed by the full-speed J. If the new host does not issue a reset within the specified time, the HNP event will be returned with an error indication.

There are two stages of the SRP, and a separate event is returned for each of them. Upon detecting a data-line pulse, the Beagle software will return an event corresponding to this condition. After detecting a data-line pulse, the software will report a $V_{BUS}$ pulse if it is seen on the bus. Note that this means that any $V_{BUS}$ pulse that occurs without a preceding data-line pulse will not be reported since it is completely out of the OTG specification. If the SRP is successful, it will be followed by a host connect event. If it is unsuccessful, then it will be followed by a host disconnect event.

### 3.4.3 Digital Inputs

Digital inputs provide a means for users to insert events into the data stream. There are four digital inputs that can be enabled individually. Whenever an enabled input changes state it will issue an event and be tagged with a timestamp of when the input was interpreted by the Beagle USB 480 analyzer. Digital inputs cannot exceed a rate of 30 MHz. Digital inputs that occur faster than that are not guaranteed to be interpreted correctly by the Beagle analyzer. Also, only one digital input event may occur per active packet. All other digital input events can only be handled after the packet has completed. Digital inputs, although guaranteed to have the correct timestamp given the previous conditions, have the possibility of being presented out of order because they are provided randomly by the user and have no direct correlation to the bus.

**Note:** the digital inputs are susceptible to cross-talk if they are not being actively driven. A situation like this could occur if a digital input has been enabled, but has not been tied to a signal. Any other nearby signal (i.e., other digital inputs or outputs) could cause the

input to activate. It is recommended that all undriven digital inputs be disabled or tied to ground.

For hardware specifications of the digital inputs refer to Section 3.4.3.

### 3.4.4 Digital Outputs

Digital outputs provide a means for users to match certain events and to send output to other devices, such as oscilloscopes. In this way, users can synchronize events on the bus with other signals they may be measuring.

Digital outputs, like digital inputs, are susceptible to cross-talk if left disabled. It is recommended that users do not attempt to use disabled digital outputs on other devices, as their characteristics are not specified. Either disconnect all connections to disabled digital outputs, or tie those outputs to ground.

There are four digital outputs that are user configurable. Each digital output has the option of being enabled, active high, or active low. Furthermore, each output can activate on specific conditions described below.

- *Digital Output 1* will match whenever the capture is running.
- *Digital Output 2* will match whenever a packet is detected on the bus.
- *Digital Output 3* will match when the selected PID, device address, and endpoint match.
- *Digital Output 4* will match when the selected PID, device address, endpoint, and data pattern match.

The digital outputs activate as soon as their match can be fully confirmed. Thus, Pins 1 and 2 will match as soon as the capture activates or rxactive goes high, respectively. However, Pins 3 and 4 must assure a match of all of their characteristics. Therefore, only once all possible PIDs, device address, and endpoints of a given packet are checked completely can the match be confirmed and the output asserted. The assertion of matched data on Pin 4 must wait until the end of the data packet to assure a match. Packets that are shorter then what is defined by the user to match will activate Pin 4 if all the data up to that point matched correctly.

Hardware specifications for the digital outputs are provided in Section 3.4.4.

### 3.4.5 Hardware Filtering

Hardware filters provide users with the ability to suppress data-less transactions. When possible, the hardware filters will discard all packets that meet the filtering criteria. These filters can save a significant amount of capture memory when used, and are highly recommended when capture-memory capacity is a concern.

Another benefit of the hardware filters is that they reduce the amount of traffic between the analysis computer and the Beagle analyzer. This is especially useful for situations where the analysis computer has a hard time keeping up with the bandwidth requirements of the Beagle analyzer. For example, the analysis computer may be running other applications or it may have other devices attached to the same bus.

There are six different hardware filters that can be used independently or in conjunction with one another. They must simply be enabled by the user. Their functionality is described below.

- *SOF Filtering* will remove all Start-of-Frame ( SOF ) tokens from the data stream. Please note that enabling the SOF filter will forfeit the ability to detect suspend and high-speed disconnects conditions on the bus.
- *IN Filtering* will attempt to remove all IN + ACK and IN + NAK pairs.
- *PING Filtering* will attempt to remove all PING + NAK pairs.
- *PRE Filtering* will remove all PRE tokens.
- *SPLIT Filtering* will attempt to remove many of the data-less SPLIT transactions. This filter will attempt to discard:

| |
|---|
| - SSPLIT + IN (for isochronous and interrupt transfers) |
| - SSPLIT + IN + ACK (for bulk and control transfers) |
| - CSPLIT + OUT + NYET |
| - CSPLIT + SETUP + NYET |
| - CSPLIT + IN + NAK |
| - CSPLIT + IN + NYET |

- *Self Filtering* will remove all packets intended for devices with the same device address as the Beagle analyzer. Due to the architecture of USB, when the Beagle analyzer is sniffing the same high-speed bus on which it is connected, it will see its own traffic on the Capture side (for more details refer to Section 1.1.2.1). This filter gives the user the opportunity to remove that traffic out of the reported data stream. This filter, however, is only effective if the Beagle USB 480 analyzer is in fact connected to the same bus as it is analyzing. If the Beagle analyzer is connected to a different host controller, this filter should be disabled, as there is a probability that another device on the Target bus will match the Beagle analyzers device address, and data to that device will be lost.

### Filters and Digital I/O

There are a couple of issues regarding the hardware filtering and digital I/O that are worth noting. Digital outputs are computed before any filtering takes place. This means

that if an output is set to activate on a normally filtered packet, the output will still activate even if the packet is never seen by the user. For example, if SOF filtering is enabled, digital outputs set to activate upon seeing an SOF PID will still activate when an SOF is on the bus.

Digital inputs can potentially invalidate a filter. The filters that are susceptible to this are the IN, PING, and SPLIT filters. These filters suppress entire transactions based on the sequence of packets on the bus. If an input trigger occurs at any time during this sequence, the entire transaction is sent to the user. As an example of this, if IN + NAK pair filtering is enabled and a digital input event occurs at any time between the start of the IN token and the very end of the NAK handshake, the entire transaction will be reported to the user. However, if no digital input event occurs, the IN + NAK pair will be discarded.

### 3.4.6 Capture Modes

The Beagle USB 480 Protocol Analyzer provides the user with three different capture modes: real-time capture, real-time capture with overflow protection, and delayed-download.

#### Real-time Capture

Real-time capture is the default capture mode. It provides the user with real-time status of the bus being monitored. The real-time capture can be stopped by three methods.

- The first method is by having the user end the capture through a `bg_disable()` call (or though the Beagle Data Center software).
- The second method is if the Beagle analyzer loses power. This is not the recommended method for stopping a capture.
- Finally, the capture will be automatically stopped by the Beagle USB 480 analyzer if the 64 MB hardware buffer (256 MB in Power models) fills to capacity. In this situation, the Beagle analyzer will no longer capture new data from the monitored bus. Instead, calls to `bg_usb480_read()` will only retrieve whatever data is remaining in the buffer. The last call of `bg_usb480_read()` will return a `BG_READ_USB_END_OF_CAPTURE` indicating that the capture has stopped and that there is no new data. The hardware buffer may fill in conditions where the analysis computer is not reading the data from the Beagle analyzer as fast as it is capturing new data.

#### Real-time Capture with Overflow Protection

Real-time Capture with Overflow Protection is essentially identical to real-time capture except that it allows for more efficient use of the hardware buffer when it nears full capacity. When the buffer is near capacity, the Beagle USB 480 analyzer will truncate all incoming packets to 4 bytes. The true length of the packet will still be reported to the

user, however only the first 4 bytes of the given packet will be returned. If the user is using a custom application, the remainder of the `packet` field will be filled with `0`s. However, all packets captured when in truncation mode will be tagged with the `BG_READ_USB_TRUNCATION_MODE` status code bit. Because packets are truncated to 4 bytes in length, only `DATA` packets have the potential of being truncated. All tokens, handshakes, etc.will still be shown in their entirety.

This mode truncates large packets reducing further usage of the hardware buffer. This allows the analysis PC a chance to siphon more data off of the Beagle analyzer before the hardware buffer becomes completely full. In other words the analysis port can catch up to the target traffic. If the buffer usage drops below a certain threshold, the analyzer will automatically return to normal operation and cease the truncation of long packets.

#### Delayed-download Capture

Delayed-download capture does not stream data to the analysis computer in real time, but instead saves all of the data in the 64 MB hardware buffer until the user is ready to download it. The size of the capture is clearly limited by the hardware buffers max capacity, so it is recommended to use the hardware filters to limit data-less transactions when appropriate.

The delayed-download capability will especially benefit those users that are analyzing high-speed traffic, but are only using a single computer with a single host controller for both the analysis computer and the target host computer. As described previously, devices on the same host controller must share the available bandwidth. Also, all high-speed devices on the same host controller will see all downstream traffic. Therefore using delayed-download will limit the Beagle analyzer's participation on the bus. In fact, if no other functions are called between the enable of the capture and the disable, there will be nearly no traffic at all between the PC and analyzer. The only traffic will be at the very start and end of the capture session.

The delayed-download will stop automatically once the buffer has reached capacity. It may also be stopped at any time by the user by calling the `bg_usb480_read` function. Polling of the status of the buffer is possible through `bg_usb480_hw_buffer_stats()`, function call. Polling the Beagle analyzer will create traffic on the bus, and thus take up some of the available bandwidth. Faster polling rates will clearly take up more bandwidth, and thus if users wish to minimize their impact on the bus, they should not poll the buffer at all. Regardless, the polling traffic itself can be filtered from the analysis data by using the hardware based Self Filter.

### 3.4.7 Match/Action System

The Beagle USB 480 Protocol Analyzer provides a simple matching system that can perform one or more actions in response to a match. The Ultimate Edition of the Beagle USB 480 Power Protocol Analyzer features Complex Matching which provides a state-based system for matching specific packet types and data patterns in addition to specific events. It also includes the Advanced trigger option, which extends the Complex

Matching framework with multiple states and extended matching facilities to build complex state machines.

#### USB 2.0 Simple Matching

The USB 2.0 simple matching system is capable of monitoring and triggering capture on USB 2.0 digital inputs, as well as asserting digital outputs and triggering capture on user-provided packet/data match patterns.

#### USB 2.0 Complex Matching

The USB 2.0 complex matching system provides a state-based facility to match specific packet types and data patterns in addition to specific events. The Ultimate Edition of the Beagle USB 480 Power Protocol Analyzers allows up to 8 states. Each state may have 4 data matching units, 1 timer match unit, and 1 asynchronous event match unit. This state-based system is used in an identical manner as the USB 3.0 Complex Matching system. See Section 3.3.8 for a description of how that is used.

### 3.4.8 Current/Voltage Monitoring

The Beagle USB 480 Power Protocol Analyzers can capture $V_{BUS}$ current and voltage readings in addition to USB data. The readings are embedded in the same data stream as the USB traffic returned by the analyzer. This functionality is tightly integrated into the Data Center Software which has been expanded to include a monitor window for real-time display and continuous tracking of the data. The monitor also offers an interactive, bi-directional way to correlate events on the $V_{BUS}$ and USB traffic.

Hardware specifications for Current/Votage Monitoring are provided in Section 2.2.5

### 3.4.9 $V_{BUS}$ Trigger

The Beagle USB 480 Power Protocol Analyzer, Ultimate Edition extends Complex Matching to include the ability to monitor $V_{BUS}$ voltage or current. Users can configure the analyzer to trigger on the rising and/or falling edge(s) of a pre-set voltage or current threshold of the $V_{BUS}$. The pre-set threshold can be included in any state of the Complex Matching state machine and each state can vary the edge(s) of the threshold it detects. It is another tool for engineers to perform complex debugging and optimize the power consumption profile of their devices.

## 3.5 Beagle $I^2C$/SPI/MDIO Protocol Analyzer Specifics

### 3.5.1 Sampling Rate

Unlike the Beagle USB analyzers, the sampling rate of the Beagle $I^2C$/SPI/MDIO analyzer is configurable. In order to accurately capture data the sampling rate must be properly set. For SPI and MDIO analysis all data lines are registered using the clock line

of the bus. The internal sampling clock is then used to retrieve the data. The sampling rate should be set to at least twice the bit rate, but preferably faster (4-5 times) if possible. Higher sampling rates can have the added benefit of increasing timing precision.

Due to the architecture of $I^2C$, there are specific bus events that occur between the standard bit-times. In order to capture these transitions, the bus must be oversampled independent of the clock line of the bus. A sampling rate of five to ten times the bus bit rate is recommended. This should not be a problem, however, since the minimum sampling rate of the Beagle $I^2C$/SPI/MDIO analyzer is 10 MHz, and $I^2C$ buses usually operate at less than 1 MHz frequencies.

The one caveat to setting the sampling rate to very high values is that higher sampling rates create more traffic on the analysis USB that connects the analyzer to the host PC. This may or may not affect performance depending on the analysis PC configuration.

# 4 Software

## 4.1 Compatibility

### 4.1.1 Overview

The Beagle software is offered as a 32-bit or a 64-bit Dynamic Linked Library (or shared object). The specific compatibility for each operating system is discussed below. Be sure the device driver has been installed before plugging in the Beagle analyzer.

### 4.1.2 Windows Compatibility

The Beagle software is compatible with Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), and Windows 7 (32-bit and 64-bit). Windows 2000 and legacy 16-bit Windows 95/98/ME operating systems are not supported.

### 4.1.3 Linux Compatibility

The Beagle software is compatible with all standard 32-bit and 64-bit distributions of Linux with kernel 2.6 and integrated USB support. When using the 32-bit library on a 64-bit distribution, the appropriate 32-bit system libraries are also required.

### 4.1.4 Mac OS X Compatibility

The Beagle software is compatible with Intel versions of Mac OS X 10.4 Tiger, 10.5 Leopard, and 10.6 Snow Leopard. Installation of the latest available update is recommended.

## 4.2 Windows USB Driver

### 4.2.1 Driver Installation

To install the appropriate USB communication driver under Windows, use the Total Phase USB Driver Installer before plugging in any device. The driver installer can be found either on the CD-ROM (use the HTML based guide that is opened when the CD is first loaded to locate the Windows installer), or in the Downloads section of the Beagle analyzer product page on the Total Phase website.

After the driver has been installed, plugging in a Beagle analyzer for the first time will cause the analyzer to be installed and associated with the correct driver. The following

steps describe the feedback the user should receive from Windows after a Beagle analyzer is plugged into a system for the first time:

**Windows XP:**

1. The Found New Hardware notification bubble will pop up from the system tray and state that the "Total Phase Beagle Protocol Analyzer" has been detected. Note that installation may take a while (30-60 seconds per device).
2. When the installation is complete, the Found New Hardware notification bubble will again pop up and state that "your new hardware is installed and ready to use."

**Windows Vista/7:**

1. A notification bubble will pop up from the system tray and state that Windows is "installing device driver software."
2. When the installation is complete, the notification bubble will again pop up and state that the "device driver software installed successfully."

To confirm that the device was correctly installed, check that the device appears in the "Device Manager." To navigate to the "Device Manager" screen, select "Control Panel | System Properties | Hardware | Device Manager" for Windows XP, or select "Control Panel | Hardware and Sound | Device Manager" for Windows Vista/7. The Beagle analyzer should appear under the "Universal Serial Bus Controllers" section for Windows XP/Vista/7.

### 4.2.2 Driver Removal

The USB communication driver can be removed from the operating system by using the Windows program removal utility. Instructions for using this utility can be found below. Alternatively, the Uninstall option found in the driver installer can also be used to remove the driver from the system.

**Note:** it is critical that all Total Phase devices have been removed from your system before removing the USB drivers.

**Windows XP:**

1. Select "Control Panel | Add or Remove Programs"
2. Select "Total Phase USB Driver" and select "Change/Remove"

3. Follow the instructions in the uninstaller

**Windows Vista/7:**

1. Select "Control Panel | Uninstall a program"
2. Right click on "Total Phase USB Driver" and select "Uninstall/Change"
3. Follow the instructions in the uninstaller

## 4.3 Linux USB Driver

The Beagle communications layer under Linux does not require a specific kernel driver to operate. However, the user must ensure independently that the libusb library is installed on the system since the Beagle library is dynamically linked to libusb.

Most modern Linux distributions use the udev subsystem to help manipulate the permissions of various system devices. This is the preferred way to support access to the Beagle analyzer such that the device is accessible by all of the users on the system upon device plug-in.

For legacy systems, there are two different ways to access the Beagle analyzer, through USB hotplug or by mounting the entire USB filesystem as world writable. Both require that `/proc/bus/usb` is mounted on the system which is the case on most standard distributions.

### 4.3.1 UDEV

Support for udev requires a single configuration file that is available on the software CD, and also listed on the Total Phase website for download. This file is `99-totalphase.rules`. Please follow the following steps to enable the appropriate permissions for the Beagle analyzer.

1. As superuser, unpack `99-totalphase.rules` to `/etc/udev/rules.d`
2. `chmod 644 /etc/udev/rules.d/99-totalphase.rules`
3. Unplug and replug your Beagle analyzer(s)

### 4.3.2 USB Hotplug

USB hotplug requires two configuration files which are available on the software CD, and also listed on the Total Phase website for download. These files are: `beagle` and `beagle.usermap`. Please follow the following steps to enable hotplugging.

1. As superuser, unpack `beagle` and `beagle.usermap` to `/etc/hotplug/usb`

2. `chmod 755 /etc/hotplug/usb/beagle`
3. `chmod 644 /etc/hotplug/usb/beagle.usermap`
4. Unplug and replug your Beagle analyzer(s)
5. Set the environment variable `USB_DEVFS_PATH` to `/proc/bus/usb`

### 4.3.3 World-Writable USB Filesystem

Finally, here is a last-ditch method for configuring your Linux system in the event that your distribution does not have udev or hotplug capabilities. The following procedure is not necessary if you were able to exercise the steps in the previous subsections.

Often, the `/proc/bus/usb` directory is mounted with read-write permissions for root and read-only permissions for all other users. If an non-privileged user wishes to use the Beagle analyzer and software, one must ensure that `/proc/bus/usb` is mounted with read-write permissions for all users. The following steps can help setup the correct permissions. Please note that these steps will make the entire USB filesystem world writable.

1. Check the current permissions by executing the following command:
   `ls -al /proc/bus/usb/001`
2. If the contents of that directory are only writable by root, proceed with the remaining steps outlined below.
3. Add the following line to the `/etc/fstab` file:

```
none /proc/bus/usb usbfs defaults,devmode=0666 0 0
```

4. Unmount the `/proc/bus/usb` directory using `umount`
5. Remount the `/proc/bus/usb` directory using `mount`
6. Repeat step 1. Now the contents of that directory should be writable by all users.
7. Set the environment variable `USB_DEVFS_PATH` to `/proc/bus/usb`

## 4.4 Mac OS X USB Driver

The Beagle communications layer under Mac OS X does not require a specific kernel driver to operate. Mac OS X 10.5 Leopard, 10.6 Snow Leopard, 10.7 Lion, and 10.8 Mountain Lion are supported. It is typically necessary to ensure that the user running the software is currently logged into the desktop. No further user configuration should be necessary.

## 4.5 USB Port Assignment

The Beagle analyzer is assigned a port on a sequential basis. The first analyzer is assigned to port 0, the second is assigned to port 1, and so on. If a Beagle analyzer is subsequently removed from the system, the remaining analyzers shift their port numbers accordingly. Hence with **n** Beagle analyzers attached, the allocated ports will be numbered from **0** to **n1**.

### 4.5.1 Detecting Ports

As described in following API documentation chapter, the `bg_find_devices` routine can be used to determine the mapping between the physical Beagle analyzers and their port numbers.

## 4.6 Beagle Dynamically Linked Library

### 4.6.1 DLL Philosophy

The Beagle DLL provides a robust approach to allow present-day Beagle-enabled applications to interoperate with future versions of the device interface software without recompilation. For example, take the case of a graphical application that is written to monitor $I^2C$, SPI, MDIO, or USB through a Beagle analyzer. At the time the program is built, the Beagle software is released as version 1.2. The Beagle interface software may be improved many months later resulting in increased performance and/or reliability; it is now released as version 1.3. The original application need not be altered or recompiled. The user can simply replace the old Beagle DLL with the newer one. How does this work? The application contains only a stub which in turn dynamically loads the DLL on the first invocation of any Beagle API function. If the DLL is replaced, the application simply loads the new one, thereby utilizing all of the improvements present in the replaced DLL.

On Linux, the DLL is technically known as a shared object (SO).

### 4.6.2 DLL Location

Total Phase provides language bindings that can be integrated into any custom application. The default behavior of locating the Beagle DLL is dependent on the operating system platform and specific programming language environment. For example, for a C or C++ application, the following rules apply:

On a Windows system, this is as follows:

1. The directory from which the application binary was loaded.
2. The applications current directory.

3. 32-bit system directory (for a 32-bit application). Examples:
   - `c:\Windows\System32` [Windows XP/Vista/7 32-bit]
   - `C:\Windows\System64` [Windows XP 64-bit]
   - `c:\Windows\SysWow64` [Windows Vista/7 64-bit]
4. 64-bit system directory (for a 64-bit application). Examples:
   - `C:\Windows\System32` [Windows XP/Vista/7 64-bit]
5. The Windows directory. (Ex: `c:\Windows` )
6. The directories listed in the PATH environment variable.

On a Linux system this is as follows:

1. First, search for the shared object in the application binary path. If the `/proc` filesystem is not present, this step is skipped.
2. Next, search in the applications current working directory.
3. Search the paths explicitly specified in `LD_LIBRARY_PATH`.
4. Finally, check any system library paths as specified in `/etc/ld.so.conf` and cached in `/etc/ld.so.cache`.

On a Mac OS X system this is as follows:

1. First, search for the shared object in the application binary path.
2. Next, search in the applications current working directory.
3. Search the paths explicitly specified in `DYLD_LIBRARY_PATH`.
4. Finally, check the /usr/lib and /usr/local/lib system library paths.

If the DLL is still not found, the `BG_UNABLE_TO_LOAD_LIBRARY` error will be returned by the binding function.

### 4.6.3 DLL Versioning

The Beagle DLL checks to ensure that the firmware of a given Beagle analyzer is compatible. Each DLL revision is tagged as being compatible with firmware revisions greater than or equal to a certain version number. Likewise, each firmware version is tagged as being compatible with DLL revisions greater than or equal to a specific version number.

Here is an example.

```
DLL v1.20: compatible with Firmware >= v1.15
Firmware v1.30: compatible with DLL >= v1.20
```

Hence, the DLL is not compatible with any firmware less than version 1.15 and the firmware is not compatible with any DLL less than version 1.20. In this example, the version number constraints are satisfied and the DLL can safely connect to the target firmware without error. If there is a version mismatch, the API calls to open the device will fail. See the API documentation for further details.

## 4.7 Rosetta Language Bindings: API Integration into Custom Applications

### 4.7.1 Overview

The Beagle Rosetta language bindings make integration of the Beagle API into custom applications simple. Accessing a Beagle analyzer's functionality simply requires function calls to the Beagle API. This API is easy to understand, much like the ANSI C library functions, (e.g., there is no unnecessary entanglement with the Windows messaging subsystem like development kits for some other embedded tools).

First, choose the Rosetta bindings appropriate for the programming language. Different Rosetta bindings are included with the software distribution on the distribution CD. They can also be found in the software download package available on the Total Phase website. Currently the following languages are supported: C/C++, Python, Visual Basic 6, Visual Basic .NET, and C#.

Next, follow the instructions for each language binding on how to integrate the bindings with your application build setup. As an example, the integration for the C language bindings is described below. (For information on how to integrate the bindings for other languages, please see the example code included on the distribution CD and also available for download on the Total Phase website.)

1. Include the `beagle.h` file included with the API software package in any C or C++ source module. The module may now use any Beagle API call listed in `beagle.h`.
2. Compile and link `beagle.c` with your application. Ensure that the include path for compilation also lists the directory in which `beagle.h` is located if the two files are not placed in the same directory.
3. Place the Beagle DLL, included with the API software package, in the same directory as the application executable or in another directory such that it will be found by the previously described search rules.

### 4.7.2 Versioning

Since a new Beagle DLL can be made available to an already compiled application, it is essential to ensure the compatibility of the Rosetta binding used by the application (e.g., `beagle.c` ) against the DLL loaded by the system. A system similar to the one employed for the DLL-Firmware cross-validation is used for the binding and DLL compatibility check.

Here is an example.

```
DLL v1.20: compatible with Binding >= v1.10
Binding v1.15: compatible with DLL >= v1.15
```

The above situation will pass the appropriate version checks. The compatibility check is performed within the binding. If there is a version mismatch, the API function will return an error code, `BG_INCOMPATIBLE_LIBRARY`.

### 4.7.3 Customizations

While provided language bindings stubs are fully functional, it is possible to modify the code found within this file according to specific requirements imposed by the application designer.

For example, in the C bindings one can modify the DLL search and loading behavior to conform to a specific paradigm. See the comments in `beagle.c` for more details.

## 4.8 Application Notes

### 4.8.1 Receive Saturation

Once enabled, the Beagle analyzer is constantly monitoring data on the target bus. Between calls to the Beagle API, these messages must be buffered somewhere in memory. This is accomplished on the analysis computer, courtesy of the operating system. Naturally the buffer is limited in size and once this buffer is full, data will be dropped. An overflow can occur when the Beagle analyzer receives data faster than the rate that it is processed – the receive link is "saturated." The system is most susceptible to saturation when monitoring large amounts of traffic over USB or high-speed SPI bus.

### 4.8.2 Threading

The Beagle DLL is designed for single-threaded environments so as to allow for maximum cross-platform compatibility. If the application design requires multi-threaded use of the Beagle analyzer's functionality, each Beagle API call can be wrapped with a thread-safe locking mechanism before and after invocation.

It is the responsibility of the application programmer to ensure that the Beagle analyzer open and close operations are thread-safe and cannot happen concurrently with any other Beagle analyzer operations. However, once a Beagle analyzer is opened, all operations to that device can be dispatched to a separate thread as long as no other threads access that same Beagle analyzer.

# 5 Firmware

## 5.1 Philosophy

The firmware included with the Beagle analyzer provides for the analysis of the supported protocols. It is installed at the factory during manufacturing. Some parts of the firmware can be updated automatically by the software. Other pieces of the firmware require a device upgrade utility. In those cases, the Beagle software automatically detects firmware compatibility and will inform the user if an upgrade is required.

## 5.2 Procedure

Firmware upgrades should be conducted using the procedure specified in the README.txt that accompanies the particular firmware revision.

# 6 API Documentation

## 6.1 Introduction

The API documentation describes the Beagle Rosetta C bindings.

## 6.2 General Data Types

The following definitions are provided for convenience. The Beagle API provides both signed and unsigned data types as well as single- and double-precision floating-point numbers.

```
typedef unsigned char         u08;
typedef unsigned short        u16;
typedef unsigned int          u32;
typedef unsigned long long    u64;
typedef signed char           s08;
typedef signed short          s16;
typedef signed int            s32;
typedef signed long long      s64;
typedef float                 f32;
typedef double                f64;
```

## 6.3 Notes on Status Codes

Most of the Beagle API functions can return a status or error code back to the caller. The complete list of status codes is provided at the end of this chapter. All of the error codes are assigned values less than 0, separating these responses from any numerical values returned by certain API functions.

Each API function can return one of two error codes with respect to the loading of the Beagle DLL, `BG_UNABLE_TO_LOAD_LIBRARY` and `BG_INCOMPATIBLE_LIBRARY`. If these status codes are received, refer to the previous sections in this datasheet that discuss the DLL and API integration of the Beagle software. Furthermore, all API calls can potentially return the errors `BG_UNABLE_TO_LOAD_DRIVER` or `BG_INCOMPATIBLE_DRIVER`. If either of these errors are seen, please make sure the driver is installed and of the correct version. Where appropriate, compare the language binding versions ( `BG_HEADER_VERSION` found in `beagle.h` and `BG_CFILE_VERSION` found in `beagle.c` ) to verify that there are no mismatches. Next, ensure that the Rosetta language binding (e.g., `beagle.c` and `beagle.h` ) are from the same release as the Beagle DLL. If all of these versions are synchronized and there are still problems, please contact Total Phase support for assistance.

Note that any API function that accepts a Beagle handle can potentially return the error code BG_INVALID_HANDLE if the handle does not correspond to a valid Beagle analyzer that has already been opened. If this error is received, check the application code to ensure that the bg_open command returned a valid handle and that this handle was not corrupted before being passed to the offending API function.

Finally, any API call that communicates with a Beagle analyzer can also return the error BG_COMMUNICATION_ERROR. This means that while the Beagle handle is valid and the communication channel is open, there was an error communicating with the device. This is possible if the device was unplugged while being used.

If either the $I^2C$, SPI, MDIO, or USB subsystems have been disabled by bg_disable, all other API functions that interact with $I^2C$, SPI, MDIO, and USB will return BG_I2C_NOT_ENABLED, BG_SPI_NOT_ENABLED, BG_MDIO_NOT_ENABLED, or BG_USB_NOT_ENABLED, respectively.

These common status responses are not reiterated for each function. Only the error codes that are specific to each API function are described below.

All of the possible error codes, along with their values and status strings, are listed following the API documentation.

## 6.4 General

### 6.4.1 Interface

#### Find Devices (bg_find_devices)

```
int bg_find_devices (int num_devices,
                     u16 * devices);
```

*Get a list of ports to which Beagle devices are attached.*

**Arguments**

| | |
|---|---|
| num_devices | maximum number of devices to return |
| devices | array into which the port numbers are returned |

**Return Value**

This function returns the number of devices found, regardless of the array size.

**Specific Error Codes**

None.

**Details**

Each element of the array is written with the port number.

Devices that are in use are ORed with BG_PORT_NOT_FREE ( 0x8000 ). Under Linux, such devices correspond to Beagle analyzers that are currently in use. Under Windows, such devices are currently in use, but it is not known if the device is a Beagle analyzer.

Example:

Devices are attached to port 0, 1, 2
ports 0 and 2 are available, and port 1 is in-use.
array => { 0x0000, 0x8001, 0x0002 }

If the input array is NULL, it is not filled with any values.

If there are more devices than the array size (as specified by nelem ), only the first nelem port numbers will be written into the array.

### Find Devices (bg_find_devices_ext)

```
int bg_find_devices_ext (int   num_devices,
                         u16 * devices,
                         int   num_ids,
                         u32 * unique_ids);
```

*Get a list of ports and unique IDs to which Beagle devices are attached.*

**Arguments**

| | |
|---|---|
| num_devices | maximum number of devices to return |
| devices | array into which the port numbers are returned |
| num_ids | maximum number of device IDs to return |
| unique_ids | array into which the unique IDs are returned |

**Return Value**

This function returns the number of devices found, regardless of the array sizes.

**Specific Error Codes**

None.

**Details**

This function is the same as `bg_find_devices()` except that it also returns the unique IDs of each Beagle device. The IDs are guaranteed to be non-zero if valid.

The IDs are the unsigned integer representation of the 10-digit serial numbers.

The number of devices and IDs returned in each of their respective arrays is determined by the minimum of `num_devices` and `num_ids`. However, if either array is `NULL`, the length passed in for the other array is used as-is, and the `NULL` array is not populated. If both arrays are `NULL`, neither array is populated, but the number of devices found is still returned.

### Open a Beagle analyzer (bg_open)

```
Beagle bg_open (int port_number);
```

*Open the Beagle port.*

**Arguments**

| | |
|---|---|
| `port_number` | The Beagle analyzer port number. This port number is the the same as the one obtained from the `bg_find_devices()` function. It is a zero-based number. |

**Return Value**

This function returns a Beagle handle, which is guaranteed to be greater than zero if valid.

**Specific Error Codes**

| | |
|---|---|
| `BG_UNABLE_TO_OPEN` | The specified port is not connected to a Beagle analyzer or the port is already in use. |
| `BG_INCOMPATIBLE_DEVICE` | There is a version mismatch between the DLL and the hardware. The DLL is not of a sufficient version for interoperability with the hardware version or vice versa. See `bg_open_ext()` in Section 6.4.1.4 for more information. |

**Details**

This function is recommended for use in simple applications where extended information is not required. For more complex applications, the use of `bg_open_ext()` is recommended.

### Open a Beagle analyzer (bg_open_ext)

```
Beagle bg_open_ext (int port_number, BeagleExt *bg_ext);
```

*Open the Beagle port, returning extended information in the supplied structure.*

#### Arguments

| | |
|---|---|
| `port_number` | same as `bg_open` |
| `bg_ext` | pointer to pre-allocated structure for extended version information available on open |

#### Return Value

This function returns a Beagle handle, which is guaranteed to be greater than zero if valid.

#### Specific Error Codes

| | |
|---|---|
| `BG_UNABLE_TO_OPEN` | The specified port is not connected to a Beagle analyzer or the port is already in use. |
| `BG_INCOMPATIBLE_DEVICE` | There is a version mismatch between the DLL and the hardware. The DLL is not of a sufficient version for interoperability with the hardware version or vice versa. The version information will be available in the memory pointed to by `bg_ext`. |

#### Details

If 0 is passed as the pointer to the structure `bg_ext`, this function will behave exactly like `bg_open()`.

The `BeagleExt` structure is described below:

```
struct BeagleExt {
    BeagleVersion version;
    /* Feature bitmap for this device. */
    int features;
};
```

The features field denotes the capabilities of the Beagle analyzer. See the API function `bg_features` for more information.

The BeagleVersion structure describes the various version dependencies of Beagle components. It can be used to determine which component caused an incompatibility error.

```
struct BeagleVersion {
  /* Software, firmware, and hardware versions. */
  u16 software;
  u16 firmware;
  u16 hardware;

  /*
  * Hardware revisions that are compatible with this software version.
  * The top 16 bits gives the maximum accepted hw revision.
  * The lower 16 bits gives the minimum accepted hw revision.
  */
  u32 hw_revs_for_sw;

  /*
  * Firmware revisions that are compatible with this software version.
  * The top 16 bits gives the maximum accepted fw revision.
  * The lower 16 bits gives the minimum accepted fw revision.
  */
  u32 fw_revs_for_sw

  /*
  * Driver revisions that are compatible with this software version.
  * The top 16 bits gives the maximum accepted driver revision.
  * The lower 16 bits gives the minimum accepted driver revision.
  * This version checking is currently only pertinent for WIN32
  * platforms.
  */
  u32 drv_revs_for_sw;

  /
* Software requires that the API must be >= this version. */
  u16 api_req_by_sw;
};
```

All version numbers are of the format:

```
(major << 8) | minor
```

example: v1.20 would be encoded as 0x0114.

The structure is zeroed before the open is attempted. It is filled with whatever information is available. For example, if the hardware version is not filled, then the device could not be queried for its version number.

This function is recommended for use in complex applications where extended information is required. For simpler applications, the use of `bg_open()` is recommended.

### Close a Beagle analyzer connection (bg_close)

```
int bg_close (Beagle beagle);
```

*Close the Beagle analyzer port.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer to be closed |

**Return Value**

The number of analyzers closed is returned on success. This will usually be 1.

**Specific Error Codes**

`None.`

**Details**

If the `handle` argument is zero, the function will attempt to close all possible handles, thereby closing all open Beagle analyzer. The total number of Beagle analyzers closed is returned by the function.

### Get Features (bg_features)

```
int bg_features (Beagle beagle);
```

*Return the device features as a bit-mask of values, or an error code if the handle is not valid.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

The features of the Beagle analyzer are returned. These are a bit-mask of the following values.

```
#define BG_FEATURE_NONE     (0)
```

```
#define BG_FEATURE_I2C      (1<<0)
#define BG_FEATURE_SPI      (1<<1)
#define BG_FEATURE_USB      (1<<2)
#define BG_FEATURE_MDIO     (1<<3)
#define BG_FEATURE_USB_HS   (1<<4)
#define BG_FEATURE_USB_SS   (1<<5)
```

**Specific Error Codes**

None.

**Details**

None.

### Get Features by Unique ID (bg_unique_id_to_features)

```
int bg_unique_id_to_features (u32 unique_id);
```

*Return the bitmask of device features for the given Beagle device, identified by `unique_id`.*

**Arguments**

| | |
|---|---|
| `beagle` | unique ID of a Beagle analyzer |

**Return Value**

The features of the Beagle analyzer are returned. See `bg_features()` for details on the bit map.

**Specific Error Codes**

None.

**Details**

None.

### Get Port (bg_port)

```
int bg_port (Beagle beagle);
```

*Return the port number for this Beagle handle.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

The port number corresponding to the given handle is returned. It is a zero-based number.

**Specific Error Codes**

None.

**Details**

None.

### Get Unique ID (bg_unique_id)

```
u32 bg_unique_id (Beagle beagle);
```

*Return the unique ID of the given Beagle analyzer.*

**Arguments**

| | |
|---|---|
| beagle | handle of a Beagle analyzer |

**Return Value**

This function returns the unique ID for this Beagle analyzer. The IDs are guaranteed to be non-zero if valid. The ID is the unsigned integer representation of the 10-digit serial number.

**Specific Error Codes**

None.

**Details**

None.

### Status String (bg_status_string)

```
const char *bg_status_string (int status);
```

*Return the status string for the given status code.*

**Arguments**

| | |
|---|---|
| status | status code returned by a Beagle API function |

**Return Value**

This function returns a human readable string that corresponds to status. If the code is not valid, it returns a NULL string.

**Specific Error Codes**

None.

**Details**

None.

### Version (bg_version)

```
int bg_version (Beagle beagle, BeagleVersion *version);
```

*Return the version matrix for the device attached to the given handle.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `version` | pointer to pre-allocated structure |

**Return Value**

A Beagle status code is returned with `BG_OK` on success.

**Specific Error Codes**

| | |
|---|---|
| `BG_COMMUNICATION_ERROR` | The firmware of the specified device can not be determined. |

**Details**

If the handle is 0 or invalid, only the software version is set.

See the details of `bg_open_ext()` for the definition of `BeagleVersion`.

### Capture Latency (bg_latency)

```
int bg_latency (Beagle beagle, u32 milliseconds);
```

*Set the capture latency to the specified number of milliseconds.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

| | |
|---|---|
| `milliseconds` | new capture latency in milliseconds |

**Return Value**

A Beagle status code is returned with `BG_OK` on success.

**Specific Error Codes**

| | |
|---|---|
| `BG_STILL_ACTIVE` | An attempt was made to change the configuration while the capture was still active. |

**Details**

Set the capture latency to the specified number of milliseconds.

The capture latency effectively splits up the total amount of buffering (as determined by `bg_host_buffer_size()` ) into smaller individual buffers. Only once one of these individual buffers is filled, does the read function return. Therefore, in order to fulfill shorter latency requirements these individual buffers are set to a smaller size. If a larger latency is requested, then the individual buffers will be set to a larger size.

Setting a small latency can increase the responsiveness of the read functions. It is important to keep in mind that there is a fixed cost to processing each individual buffer that is independent of buffer size. Therefore, the trade-off is that using a small latency will increase the overhead *per byte* buffered. A large latency setting decreases that overhead, but increases the amount of time that the library must wait for each buffer to fill before the library can process their contents.

This setting is distinctly different than the timeout setting. The latency time should be set to a value shorter than the timeout time.

### Timeout Value (bg_timeout)

```
int bg_timeout (Beagle beagle, u32 milliseconds);
```

*Set the read timeout to the specified number of milliseconds.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `milliseconds` | new timeout value in milliseconds |

**Return Value**

A Beagle status code is returned with `BG_OK` on success.

**Specific Error Codes**

None.

**Details**

Set the idle timeout to the specified number of milliseconds.

This function sets the amount of time that the read functions will wait before returning if the bus is idle. If a read function is called and there has been no new data on the bus for the specified timeout interval, the function will return with the BG_READ_TIMEOUT flag of the status value set and the return value will indicate the number of bytes of data that the Beagle analyzer was able to capture prior to the timeout.

If the timeout is set to 0, there is no timeout interval and the read functions will block until the requested amount of data is captured or a complete packet with the appropriate bus end condition is observed.

This setting is distinctly different than the latency setting. The timeout time should be set to a value longer than the latency time.

### Sleep (bg_sleep_ms)

```
u32 bg_sleep_ms (u32 milliseconds);
```

*Sleep for given amount of time.*

**Arguments**

| | |
|---|---|
| milliseconds | number of milliseconds to sleep |

**Return Value**

This function returns the number of milliseconds slept.

**Specific Error Codes**

None.

**Details**

This function provides a convenient cross-platform function to sleep the current thread using standard operating system functions.

The accuracy of this function depends on the operating system scheduler. This function will return the number of milliseconds that were actually slept.

### Target Power (bg_target_power)

```
int bg_target_power (Beagle beagle, u08 power_flag);
```

*Activate/deactivate target power pins 4 and 6.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `power_mask` | enumerated values specifying power pin state. See Table 7. |

**Table 7** : Power Flag definitions

| | |
|---|---|
| `BG_TARGET_POWER_OFF` | Disable target power pin |
| `BG_TARGET_POWER_ON` | Enable target power pin |
| `BG_TARGET_POWER_QUERY` | Queries the target power pin state |

#### Return Value

The current state of the target power pins on the Beagle analyzer will be returned. The configuration will be described by the same values as in the table above.

#### Specific Error Codes

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | The hardware version is not compatible with this feature. Only the Beagle $I^2C$/SPI/MDIO monitor supports switchable target power pins. |

#### Details

This function is only available on the Beagle $I^2C$/SPI/MDIO Protocol Analyzer.

Both target power pins are controlled together. Independent control is not supported. This function may be executed in any operation mode.

For the most part, target power should be left off, as the Beagle analyzer is normally passively monitoring the bus.

### Host Interface Speed (bg_host_ifce_speed)

```
int bg_host_ifce_speed (Beagle beagle);
```

*Query the host interface speed.*

**Arguments**

`beagle` handle of a Beagle analyzer

**Return Value**

This function returns enumerated values specifying the USB speed at which the analysis computer is communicating with the given Beagle analyzer. See Table 8.

**Table 8** : Interface Speed definitions

| | |
|---|---|
| `BG_HOST_IFCE_FULL_SPEED` | Full-speed (12 Mbps) interface |
| `BG_HOST_IFCE_HIGH_SPEED` | High-speed (480 Mbps) interface |

**Specific Error Codes**

`None.`

**Details**

Used to determine the USB communication rate between the Beagle analyzer and the analysis PC. The Beagle analyzers require a high-speed USB connection with the host. Capturing from a Beagle analyzer that is connected at full-speed can cause data to be lost and corruption of capture data.

### 6.4.2 Buffering

#### Host Buffer Size (bg_host_buffer_size)

```
int bg_host_buffer_size(Beagle beagle, u32 size_bytes);
```

*Set the amount of buffering that is to be allocated on the analysis PC*

**Arguments**

`beagle` handle of a Beagle analyzer
`num_bytes` number of bytes in buffer

**Return Value**

This function returns the actual amount of buffering set.

**Specific Error Codes**

| | |
|---|---|
| BG_STILL_ACTIVE | An attempt was made to change the configuration while the capture was still active. |

**Details**

This function sets the amount of memory allocated to buffering data that has been siphoned off the Beagle analyzer by the host software library, but not yet read by the application. The absolute minimum and maximum values for this buffer size are 64 kB and 16 MB, respectively. The requested buffer size is matched as closely as possible by the function, and the function will keep the actual buffer size within these boundaries. For example, if 32 kB of buffering is requested, then 64 kB will actually be set.

If `num_bytes` is 0, the function will return the amount of buffering currently set on the PC and will leave the amount of buffering unmodified. This function can be called in this fashion even when the capture is active as it does not attempt to change the configuration. It is important to note that `bg_latency()` and `bg_sample_rate()` can have an effect on the total buffer size. Therefore, to accurately determine how much buffering has been set on the PC, this call should be made after all the configurations have been set.

If the application does not read data from the software library quickly enough, the entire host-side buffer will fill. For most of the Beagle analyzers this means that any new traffic on the target bus will be dropped. The Beagle USB 480 analyzer, however, has a large on-board memory buffer to solve this issue. To understand the operation of the Beagle USB 480 analyzer and how it relates to the API, please refer to Section 6.8.

### Available Read Buffering (bg_host_buffer_free)

```
int bg_host_buffer_free (Beagle beagle);
```

*Query the amount of read buffering available.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

The amount of available USB read buffering in bytes.

**Specific Error Codes**

`None.`

**Details**

USB read buffers are used by the analysis computer to receive the incoming data from the Beagle analyzer. Calling this function will return the amount of PC buffering available to receive data as of the last `bg_*_read()` call. If the amount of available USB buffering drops close to zero, capture data from the device may be lost.

### Used Read Buffering (bg_host_buffer_used)

```
int bg_host_buffer_used (Beagle beagle);
```

*Query the amount of used USB read buffering.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer. |

**Return Value**

The amount of used USB read buffering in bytes.

**Specific Error Codes**

`None.`

**Details**

USB read buffers are used by the analysis computer to receive the incoming data from the Beagle analyzer. Calling this function will return the amount of PC buffering filled with received data as of the last `bg_*_read()` call. If the amount of used USB buffering comes close to the total buffer size, capture data from the device may be lost.

### Communication Speed Benchmark (bg_commtest)

```
int bg_commtest (Beagle beagle, int num_samples, int delay_count);
```

*Test the Beagle analyzer communication link performance.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `num_samples` | number of samples to receive from the analyzer. |
| `delay_count` | count delay on the host before processing each sample |

**Return Value**

The number of communication errors received during the test.

#### Specific Error Codes

None.

#### Details

This function tests the host computers ability to process data received from the Beagle analyzer. The function commands the given Beagle analyzer to send test packets at the given frequency (see `bg_samplerate()` ) to the host computer over the USB interface. The `delay_count` variable provides a way for the application programmer to add an artificial counter delay between each sample processed by the host. For large delay values, it will be harder for the host to keep up with the data rate over the USB bus, thereby leading to more communication errors.

### 6.4.3 Monitoring API

#### Enable Monitoring (bg_enable)

```
int bg_enable (Beagle beagle, BeagleProtocol protocol);
```

*Start monitoring packets on the selected interface.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `protocol` | enumerated values specifying the protocol to monitor (see Table 9 ) |

**Table 9** : `BeagleProtocol` enumerated values

| | |
|---|---|
| BG_PROTOCOL_NONE | No Protocol |
| BG_PROTOCOL_COMMTEST | Comm Tester |
| BG_PROTOCOL_USB | USB Protocol |
| BG_PROTOCOL_I2C | $I^2C$ Protocol |
| BG_PROTOCOL_SPI | SPI Protocol |
| BG_PROTOCOL_MDIO | MDIO Protocol |

#### Return Value

A Beagle status code of BG_OK is returned on success.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | The connected Beagle analyzer does not support capturing for the requested protocol. |
| `BG_UNKNOWN_PROTOCOL` | A protocol was requested that does not appear in the enumeration detailed in Table 9. |

**Details**

This function enables monitoring on the given Beagle analyzer. See the section on the protocol-specific APIs. Functions for retrieving the capture data from the Beagle analyzer are described therein.

### Stop Monitoring (bg_disable)

```
int bg_disable (Beagle beagle);
```

*Stop monitoring of packets.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

A Beagle status code of `BG_OK` is returned on success.

**Specific Error Codes**

`None.`

**Details**

Stops monitoring on the given Beagle analyzer.

### Sample Rate (bg_samplerate)

```
int bg_samplerate (Beagle beagle, int samplerate_khz);
```

*Set the sample rate in kilohertz.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `samplerate_khz` | New sample rate in kilohertz |

**Return Value**

This function returns the actual sample rate set.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | The Beagle analyzer does not support changing the sample rate. |
| `BG_STILL_ACTIVE` | An attempt was made to change the configuration while the capture was still active. |

**Details**

Changes the sample rate for a Beagle analyzer. The device must not currently have monitoring enabled.

If `samplerate_khz` is 0, the function will return the sample rate currently set on the Beagle analyzer and the sample rate will be left unmodified. The Beagle USB 12 analyzer and the Beagle USB 480 analyzer do not support changing the sample rate, so it will always return the current sample rate.

### Bit Timing Size (bg_bit_timing_size)

```
int bg_bit_timing_size (BeagleProtocol  protocol,
                        int             num_data_bytes);
```

*Get the size of the timing data for the given protocol and data size.*

**Arguments**

| | |
|---|---|
| `protocol` | enumerated values specifying the protocol of the data (see Table 9 ) |
| `num_data_bytes` | number of data bytes expected |

**Return Value**

The number of timing entries to expect for given number of data bytes for the given protocol.

**Specific Error Codes**

`None.`

**Details**

Call this function before calling the `bg_***_read_bit_timing()` API functions to determine how large a `bit_timing` array to allocate.

For `BG_PROTOCOL_MDIO`, this function will always return the value 32, regardless of the the value passed for `num_data_bytes`.

### Trigger Capture (bg_capture_trigger)

```
int bg_capture_trigger (Beagle beagle);
```

*Trigger the capture.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is supported only for analyzers with on-board triggering capability.

Calling this function triggers the capture. Once the capture has been triggered, data can be downloaded from the hardware buffer by calling the read function.

### Wait for Capture to Trigger (bg_capture_trigger_wait)

```
int bg_capture_trigger_wait (Beagle                beagle
                             u32                   timeout_ms
                             BeagleCaptureStatus * status);
```

*Wait for capture to trigger.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `timeout_ms` | timeout value |
| `status` | filled with enumerated value described in Table 10 |

**Table 10** : BeagleCaptureStatus Enums

| | |
|---|---|
| BG_CAPTURE_STATUS_INACTIVE | Capture is not active |
| BG_CAPTURE_STATUS_SYNC_STANDBY | Waiting for capture to start on all analyzers connected by Cross-Analyzer Sync |
| BG_CAPTURE_STATUS_PRE_TRIGGER | Filling pre-trigger |
| BG_CAPTURE_STATUS_PRE_TRIGGER_SYNC | Synchronizing timestampes between multiple streams |
| BG_CAPTURE_STATUS_POST_TRIGGER | Filling post-trigger |
| BG_CAPTURE_STATUS_TRANSFER | Capture stopped, downloading data |
| BG_CAPTURE_STATUS_COMPLETE | Capture stopped, all data downloaded |

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

BG_FUNCTION_NOT_AVAILABLE Function not supported by device.

**Details**

This function is supported only for analyzers with on-board triggering capability.

This function will block while the capture is in the pre-trigger or sync-standby states or until timeout_ms milliseconds have passed.

### Abort Capture (bg_capture_stop)

```
int bg_capture_stop (Beagle beagle);
```

*Stop capturing data.*

**Arguments**

beagle handle of a Beagle analyzer

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| BG_FUNCTION_NOT_AVAILABLE | Function not supported by device. |

**Details**

This function is supported only for analyzers with on-board triggering capability. For other analyzers, use `bg_disable` to stop the capture and download to PC.

Captured data may still be read from the hardware buffer after calling this function, but new data will not be monitored. This is different that `bg_disable`, which discards the capture buffer.

After calling this function, `bg_disable` must be called before a new capture can be enabled.

## 6.5 Notes on Protocol-Specific Read Functions

All read functions return a status value through the `status` parameter. Table 11 provides a listing of all the status codes that are shared throughout all the protocols.

**Table 11** : Read Status definitions

| | |
|---|---|
| BG_READ_OK | Read successful. |
| BG_READ_TIMEOUT | No data was seen before the timeout interval occurred. This may indicate that no data was seen on the bus or there was a pause in the transmission of data longer than the timeout interval. |
| BG_READ_ERR_MIDDLE_OF_PACKET | Data collection was started in the middle of a packet. This indicates that a transaction was already being transmitted across the bus when the read function was called. |
| BG_READ_ERR_SHORT_BUFFER | The packet was longer than the buffer size. The buffer passed to the read function was too short to contain the full data of the transaction. |
| BG_READ_ERR_PARTIAL_LAST_BYTE | The last byte in the buffer is incomplete. The number of bits of data captured did not align to the expected data size. For example, for $I^2C$ the number of bits received was not a multiple of 9 (8 data bits plus 1 ACK/NACK bit). |
| BG_READ_ERR_UNEXPECTED | An unexpected event occurred on the bus. The event is still presented to the user, however it is tagged with this status flag. |

# 6.6 I²C API

## 6.6.1 Notes

The I²C API functions are only available for the Beagle I²C/SPI/MDIO Protocol Analyzer.

## 6.6.2 I²C Monitor Interface

### I²C Pullups (bg_i2c_pullup)

```
int bg_i2c_pullup (Beagle   beagle,
                   u08      pullup_flag);
```

*Enables, disables and queries the I²C pullup resistors.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `pullup_flag` | the function to perform as detailed in Table 12 |

**Table 12** : Pullup definitions

| | |
|---|---|
| `BG_I2C_PULLUP_OFF` | Disable the pullup resistors. |
| `BG_I2C_PULLUP_ON` | Enable the pullup resistors. |
| `BG_I2C_PULLUP_QUERY` | Query the status of the pullup resistors. |

**Return Value**

A Beagle status code of `BG_OK` is returned on success. If the value passed for `pullup_flag` is BG_I2C_PULLUP_QUERY, the state of the pullups is returned.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | The hardware version is not compatible with this feature. Only I²C devices support switchable pullup pins. |

**Details**

Sets and queries the state of the pullup resistors on the I²C lines. Normally the pullups will be set by the host and target devices, so this function will not be used.

### Read I²C (bg_i2c_read)

```
int bg_i2c_read (Beagle    beagle,
                 u32 *     status,
                 u64 *     time_sop,
                 u64 *     time_duration,
                 u32 *     time_dataoffset,
                 int       max_bytes,
                 u16 *     data_in);
```

*Read packet from the I²C port.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `status` | filled with the status bitmask as detailed in Tables 11 and 13 |
| `time_sop` | filled with the timestamp when the packet begins |
| `time_duration` | filled with the number of ticks that it took to read the data |
| `time_dataoffset` | filled with the timestamp when data appeared on the bus |
| `max_bytes` | maximum number of bytes to read |
| `data_in` | an allocated array of `u16` which is filled with the received data |

**Table 13** : I²C Specific Read Status definitions

| | |
|---|---|
| `BG_READ_I2C_NO_STOP` | The I²C stop condition was not observed on the bus. This can be caused either by a read timeout or by a I²C repeated start condition. |

**Return Value**

This function returns the number of bytes read or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

The function will block until the requested amount of data is captured, a complete packet with a stop or repeated start condition is observed, or the bus is idle for longer than the timeout interval set. See Section 6.4.1.12 for information on the `bg_latency()` and `bg_timeout()` functions which affect the behavior of this function.

For each u16 written to `data_in` by the function, the lower 8-bits represent the value of a byte of data sent across the bus and bit 8 represents the ACK or NACK value for that byte. A 0 in bit 8 represents an ACK and a 1 in bit 8 represents a NACK. See Table 14 for constants that may be used as bit mask to access the appropriate fields in `data_in`.

All of the timing data is measured in ticks of the sample rate clock.

**Table 14** : I²C Data Mask constants

| Constant name | Value | Description |
|---|---|---|
| BG_I2C_MONITOR_DATA | 0x00ff | Mask to access data field. |
| BG_I2C_MONITOR_NACK | 0x0100 | Mask to access ACK/NACK field. |

The `data_in` pointer should be allocated at least as large as `max_bytes`.

All of the timing data is measured in ticks of the sample clock.

### Read I²C with data-level timing (bg_i2c_read_data_timing)

```
int bg_i2c_read_data_timing (Beagle     beagle,
                             u32 *      status,
                             u64 *      time_sop,
                             u64 *      time_duration,
                             u32 *      time_dataoffset,
                             int        max_bytes,
                             u16 *      data_in,
                             int        max_timing,
                             u32 *      data_timing);
```

*Read data from the I²C port.*

**Arguments**

| | |
|---|---|
| `common_args` | see `bg_i2c_read()` for common arguments |
| `max_timing` | size of `data_timing` array |

| | |
|---|---|
| `data_timing` | an allocated array of u32 which is filled with timing data for each data word read |

**Return Value**

This function returns the number of bytes read or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

This function is an extension of the `bg_i2c_read()` function with the added feature of giving data-level timing. All of the `bg_i2c_read()` arguments and details apply.

The values in the `data_timing` array give the offset of the start of each data word from `time_sop`. A data word includes all 8 bits of data as well as the acknowledgment bit.

The `data_timing` array should be allocated at least as large as `max_timing`.

### Read I²C with bit-level timing (bg_i2c_read_bit_timing)

```
int bg_i2c_read_bit_timing (Beagle   beagle,
                            u32 *    status,
                            u64 *    time_sop,
                            u64 *    time_duration,
                            u32 *    time_dataoffset,
                            int      max_bytes,
                            u16 *    data_in,
                            int      max_timing,
                            u32 *    bit_timing);
```

*Read data from the I²C port.*

**Arguments**

| | |
|---|---|
| `common_args` | see `bg_i2c_read()` for common arguments |
| `max_timing` | size of `bit_timing` array |
| `bit_timing` | an allocated array of u32 which is filled with the timing data for each bit read |

**Return Value**

This function returns the number of bytes read or a negative value indicating an error.

**Specific Error Codes**

None.

**Details**

This function is an extension of the `bg_i2c_read()` function with the added feature of giving bit-level timing. All of the `bg_i2c_read()` arguments and details apply.

The values in the `bit_timing` array give the offset of each bit from `time_sop`.

The `bit_timing` array should be allocated at least as large as `max_timing`. Use the function `bg_bit_timing_size()` (in Section 6.4.3.4 ) to determine how large an array to allocate for `bit_timing`.

# 6.7 SPI API

## 6.7.1 Notes

The SPI API functions are only available for the Beagle $I^2C$/SPI/MDIO Protocol Analyzer.

## 6.7.2 SPI Monitor Interface

### SPI Configuration (bg_spi_configure)

```
int bg_spi_configure (Beagle                    beagle,
                      BeagleSpiSSPolarity       ss_polarity,
                      BeagleSpiSckSamplingEdge  sck_sampling_edge,
                      BeagleSpiBitorder         bitorder);
```

*Sets SPI bus parameters.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `ss_polarity` | sets the slave select detection to active-low or active-high bit polarity, see Table 15 |

| | |
|---|---|
| `sck_sampling_edge` | sets data sampling on the leading or trailing edge of the clock signal, see Table 16 |
| `bitorder` | sets big-endian or little-endian bit order, see Table 17 |

**Table 15** : SPI SS Polarity definitions

| | |
|---|---|
| `BG_SPI_SS_ACTIVE_LOW` | Set active low polarity |
| `BG_SPI_SS_ACTIVE_HIGH` | Set active high polarity |

**Table 16** : SPI SCK Sampling Edge definitions

| | |
|---|---|
| `BG_SPI_SCK_SAMPLING_EDGE_RISING` | Sample on the leading edge |
| `BG_SPI_SCK_SAMPLING_EDGE_FALLING` | Sample on the trailing edge |

**Table 17** : SPI Bit Order definitions

| | |
|---|---|
| `BG_SPI_BITORDER_MSB` | Big-endian bit ordering |
| `BG_SPI_BITORDER_LSB` | Little-endian bit ordering |

### Return Value

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

### Specific Error Codes

| | |
|---|---|
| `BG_STILL_ACTIVE` | An attempt was made to change the configuration while the capture was still active. |
| `BG_FUNCTION_NOT_AVAILABLE` | The hardware version is not compatible with this feature. Only the $I^2C$/SPI/MDIO device supports SPI configuration. |

### Details

The SPI standard is much more loosely defined than $I^2C$, MDIO, or USB. As a consequence, the SPI monitor must be configured to match the parameters of the device being monitored. If the configuration of the SPI monitor does not match the

configuration of the SPI devices being monitored, the capture data from the monitor may be corrupted.

### Read SPI (bg_spi_read)

```
int bg_spi_read (Beagle   beagle,
                 u32 *    status,
                 u64 *    time_sop,
                 u64 *    time_duration,
                 u32 *    time_dataoffset,
                 int      mosi_max_bytes,
                 u08 *    data_mosi,
                 int      miso_max_bytes,
                 u08 *    data_miso);
```

*Read data from the SPI port.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `status` | filled with the status bitmask as detailed in Table 11 |
| `time_sop` | filled with the timestamp when the data read begins |
| `time_duration` | filled with the number of ticks that it took to read the data |
| `time_dataoffset` | filled with the timestamp when data appeared on the bus |
| `mosi_max_bytes` | maximum number of MOSI bytes to fill |
| `data_mosi` | an allocated array of `u08` which is filled with the data sent from the master to the slave |
| `miso_max_bytes` | maximum number of MISO bytes to fill |
| `data_miso` | an allocated array of `u08` which is filled with the data sent from the slave to the master |

#### Return Value

This function returns the number of bytes read or a negative value indicating an error.

#### Specific Error Codes

`None.`

#### Details

The function will block until the requested amount of data is captured, a complete packet with slave select deassertion is observed, or the bus is idle for longer than the timeout interval set. See Section 6.4.1.12 for information on the `bg_latency()` and `bg_timeout()` functions which affect the behavior of this function.

The `data_mosi` array should be allocated at least as large as `mosi_max_bytes`. The `data_miso` array should be allocated at least as large as `miso_max_bytes`.

All of the timing data is measured in ticks of the sample clock.

### Read SPI with data-level timing (bg_spi_read_data_timing)

```
int bg_spi_read_data_timing (Beagle   beagle,
                             u32 *    status,
                             u64 *    time_sop,
                             u64 *    time_duration,
                             u32 *    time_dataoffset,
                             int      mosi_max_bytes,
                             u08 *    data_mosi,
                             int      miso_max_bytes,
                             u08 *    data_miso,
                             int      max_timing,
                             u32 *    data_timing);
```

*Read data from the SPI port.*

#### Arguments

| | |
|---|---|
| `common_args` | see `bg_spi_read()` for common arguments |
| `max_timing` | size of `data_timing` array |
| `data_timing` | an allocated array of u32 which is filled with timing data for each data word read |

#### Return Value

This function returns the number of bytes read or a negative value indicating an error.

#### Specific Error Codes

None.

#### Details

This function is an extension of the `bg_spi_read()` function with the added feature of byte-level timing. All of the `bg_spi_read()` arguments and details apply.

The values in the `data_timing` array give the offset of the start of each data word from `time_sop`. For SPI, a data word is considered a single byte.

The `data_timing` array should be allocated at least as large as `max_timing`.

### Read SPI with bit-level timing (bg_spi_read_bit_timing)

```
int bg_spi_read_bit_timing (Beagle   beagle,
                            u32 *    status,
                            u64 *    time_sop,
                            u64 *    time_duration,
                            u32 *    time_dataoffset,
                            int      mosi_max_bytes,
                            u08 *    data_mosi,
                            int      miso_max_bytes,
                            u08 *    data_miso,
                            int      max_timing,
                            u32 *    bit_timing);
```

*Read data from the SPI port.*

#### Arguments

| | |
|---|---|
| `common_args` | see `bg_spi_read()` for common arguments |
| `max_timing` | size of `bit_timing` array |
| `bit_timing` | an allocated array of `u32` which is filled with the timing data for each bit read |

#### Return Value

This function returns the number of bytes read or a negative value indicating an error.

#### Specific Error Codes

`None.`

#### Details

This function is an extension of the `bg_spi_read()` function with the added feature of bit-level timing. All of the `bg_spi_read()` arguments and details apply.

The values in the `bit_timing` array give the offset of each bit from `time_sop`.

The `bit_timing` array should be allocated at least as large as `max_timing`. Use the function `bg_bit_timing_size()` (in Section 6.4.3.4) to determine how large an array to allocate for `bit_timing`.

## 6.8 USB API

### 6.8.1 Notes

1. All USB functions can be used with any Beagle USB analyzer, but an error code will be returned if the analyzer's hardware does not support the required functionality.
2. The following functionality is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers:
    1. Hardware-based USB statistics system
    2. Licensing
    3. Memory tests
3. The following functionality is not supported for the Beagle USB 12 and the non-Power model of Beagle USB 480 Protocol Analyzers:
    1. On-board triggering capability
    2. Controlling $V_{BUS}$ provided to target device
4. The following functionality is not supported for the Beagle USB 12 Protocol Analyzers:
    1. Configuring USB 2.0 target speed
    2. Digital I/O
    3. Hardware Filters
5. Delayed-download capture mode is supported only for the Beagle USB 480 Protocol Analyzer.
6. Reading data with data-level timing and bit-level timing is supported only for the Beagle USB 12 Protocol Analyzers.

7. The USB 3.0 API functions are supported only for the Beagle USB 5000 SuperSpeed Protocol Analyzers.

8. The USB 5000 API functions are supported only for the Beagle USB 5000 SuperSpeed Protocol Analyzers.

9. The first byte of a captured USB 2.0 packet is the packet ID (PID). An enumeration is provided that defines all the possible PIDs which is listed in Table 18.

**Table 18** : USB 2.0 Packet ID definitions

| | |
|---|---|
| BG_USB_PID_OUT | 0xe1 |
| BG_USB_PID_IN | 0x69 |
| BG_USB_PID_SOF | 0xa5 |
| BG_USB_PID_SETUP | 0x2d |
| BG_USB_PID_DATA0 | 0xc3 |
| BG_USB_PID_DATA1 | 0x4b |
| BG_USB_PID_DATA2 | 0x87 |
| BG_USB_PID_MDATA | 0x0f |
| BG_USB_PID_ACK | 0xd2 |
| BG_USB_PID_NAK | 0x5a |
| BG_USB_PID_STALL | 0x1e |
| BG_USB_PID_NYET | 0x96 |
| BG_USB_PID_PRE | 0x3c |
| BG_USB_PID_ERR | 0x3c |
| BG_USB_PID_SPLIT | 0x78 |
| BG_USB_PID_PING | 0xb4 |
| BG_USB_PID_EXT | 0xf0 |

10. In addition to the general read status values in Table 11, the USB read functions can also return USB specific status values. The enumerated types are listed in Table 19.

**Table 19** : USB Read Status definitions

| **Status Codes for USB 12, USB 480, and USB 5000** | |
|---|---|
| BG_READ_USB_ERR_BAD_SIGNALS | Incorrect line states |
| BG_READ_USB_ERR_BAD_PID | Captured packet has bad PID |

| | |
|---|---|
| BG_READ_USB_ERR_BAD_CRC | Captured packet has bad CRC |
| **USB 12 Specific Status Codes** | |
| BG_READ_USB_ERR_BAD_SYNC | Cannot find SYNC signal |
| BG_READ_USB_ERR_BIT_STUFF | Bit stuffing error detected |
| BG_READ_USB_ERR_FALSE_EOP | Incorrect End of packet |
| BG_READ_USB_ERR_LONG_EOP | End of packet too long |
| **Status Codes for USB 480 and USB 5000** | |
| BG_READ_USB_TRUNCATION_LEN_MASK | Available truncated data mask |
| BG_READ_USB_TRUNCATION_MODE | Captured packet in truncation mode |
| BG_READ_USB_END_OF_CAPTURE | Capture has ended |
| **USB 5000 Specific Status Codes** | |
| BG_READ_USB_ERR_BAD_SLC_CRC_1 | CRC error in 1st SLC |
| BG_READ_USB_ERR_BAD_SLC_CRC_2 | CRC error in 2nd SLC |
| BG_READ_USB_ERR_BAD_SHP_CRC_16 | CRC-16 error in SHP |
| BG_READ_USB_ERR_BAD_SHP_CRC_5 | CRC-5 error in SHP |
| BG_READ_USB_ERR_BAD_SDP_CRC | CRC error in SDP |
| BG_READ_USB_EDB_FRAMING | EDB end frame in SDP |
| BG_READ_ERR_UNK_END_OF_FRAME | Unknown end of frame |
| BG_READ_ERR_DATA_LEN_INVALID | Data length invalid |
| BG_READ_USB_PKT_TYPE_LINK | Link packet |
| BG_READ_USB_PKT_TYPE_HDR | Header packet |
| BG_READ_USB_PKT_TYPE_DP | Data packet |
| BG_READ_USB_PKT_TYPE_TSEQ | TSEQ packet |
| BG_READ_USB_PKT_TYPE_TS1 | TS1 packet |
| BG_READ_USB_PKT_TYPE_TS2 | TS2 packet |
| BG_READ_USB_ERR_BAD_TS | Error in training ordered set |
| BG_READ_USB_ERR_FRAMING | 1 symbol corruption in framing |

11. Additional event information is returned by the USB read functions through the `events` argument. The event information is bitmask encoded with the enumerated types defined in Tables 20, 21, 22, 24, 25, 23. Refer to Section 1.1.2 for details on how these events pertain to the USB architecture.

**Table 20** : USB 12, 480 and 5000 Event Code definitions

| **Event Codes for USB 12, USB 480, and USB 5000** |
|---|

| | |
|---|---|
| BG_EVENT_USB_HOST_DISCONNECT | Target Host disconnected |
| BG_EVENT_USB_TARGET_DISCONNECT | Target Device disconnected |
| BG_EVENT_USB_HOST_CONNECT | Target Host connected |
| BG_EVENT_USB_TARGET_CONNECT | Target Device connected |
| BG_EVENT_USB_RESET | Bus put into reset state |

**Table 21** : USB 480 and 5000 USB 2.0 Event Code definitions

| **Event Codes for USB 480 and USB 5000** | |
|---|---|
| BG_EVENT_USB_J_CHIRP | Chirp-J detected |
| BG_EVENT_USB_K_CHIRP | Chirp-K detected |
| BG_EVENT_USB_SPEED_UNKNOWN | Communication speed is unknown |
| BG_EVENT_USB_LOW_SPEED | Low-speed bus operation detected |
| BG_EVENT_USB_FULL_SPEED | Full-speed bus operation detected |
| BG_EVENT_USB_HIGH_SPEED | High-speed bus operation detected |
| BG_EVENT_USB_LOW_OVER_FULL_SPEED | Low-over-full-speed bus operation detected |
| BG_EVENT_USB_SUSPEND | Bus has entered suspend state |
| BG_EVENT_USB_RESUME | Bus has left suspend state |
| BG_EVENT_USB_KEEP_ALIVE | Low-speed keep-alive detected |
| BG_EVENT_USB_OTG_HNP | OTG HNP detected |
| BG_EVENT_USB_OTG_SRP_DATA_PULSE | OTG SRP data-line pulse detected |
| BG_EVENT_USB_OTG_SRP_VBUS_PULSE | OTG SRP $V_{BUS}$ pulse detected |
| BG_EVENT_USB_DIGITAL_INPUT | One or more digital inputs have changed state |
| BG_EVENT_USB_DIGITAL_INPUT_MASK | Bitmask of line state for each input pin |

**Table 22** : USB 5000 USB 2.0 Event Code definitions

| Event Codes for USB 5000 | |
|---|---|
| BG_EVENT_USB_SMA_EXTIN_DETECTED | External Input change detected |
| BG_EVENT_USB_CHIRP_DETECTED | Chrip detected |

12. Manual triggers are represented by a lone BG_EVENT_USB_TRIGGER event (no data or other events).

13. Simple triggers are represented by a BG_EVENT_USB_TRIGGER event, along with the triggering packet or event.

14. Complex triggers are represented by a BG_EVENT_COMPLEX_TRIGGER event along with state information indicating the complex state in which the trigger occurred. Complex triggers, like simple triggers, come with the triggering packet or event.

**Table 23** : USB 5000 USB 2.0 and 3.0 Trigger Event Code definitions

| Trigger Event Codes for USB 5000 | |
|---|---|
| BG_EVENT_USB_VBUS_TRIGGER | $V_{BUS}$ trigger event detected |
| BG_EVENT_USB_COMPLEX_TIMER | Complex timer lapsed |
| BG_EVENT_USB_COMPLEX_TRIGGER | Complex trigger event |
| BG_EVENT_USB_TRIGGER | Simple trigger event |
| BG_EVENT_USB_TRIGGER_STATE_MASK | Bitmask of trigger state number |
| BG_EVENT_USB_TRIGGER_STATE_SHIFT | Shift amount to recover state number |
| BG_EVENT_USB_TRIGGER_STATE_0 | Trigger state 0 |
| BG_EVENT_USB_TRIGGER_STATE_1 | Trigger state 1 |
| BG_EVENT_USB_TRIGGER_STATE_2 | Trigger state 2 |
| BG_EVENT_USB_TRIGGER_STATE_3 | Trigger state 3 |
| BG_EVENT_USB_TRIGGER_STATE_4 | Trigger state 4 |
| BG_EVENT_USB_TRIGGER_STATE_5 | Trigger state 5 |
| BG_EVENT_USB_TRIGGER_STATE_6 | Trigger state 6 |
| BG_EVENT_USB_TRIGGER_STATE_7 | Trigger state 7 |

**Table 24** : USB 5000 USB 3.0 General Event Code definitions

| Event Codes for USB 5000 | |
|---|---|
| BG_EVENT_USB_LFPS | LFPS event |
| BG_EVENT_USB_LTSSM | LTSSM event |
| BG_EVENT_USB_VBUS_PRESENT | Host $V_{BUS}$ present |
| BG_EVENT_USB_VBUS_ABSENT | Host $V_{BUS}$ absent |
| BG_EVENT_USB_SCRAMBLING_ENABLED | USB scrambling enabled |
| BG_EVENT_USB_SCRAMBLING_DISABLED | USB scrambling disabled |
| BG_EVENT_USB_POLARITY_NORMAL | Normal lane polarity |
| BG_EVENT_USB_POLARITY_REVERSED | Lane polarity inverted |
| BG_EVENT_USB_PHY_ERROR | PHY error |
| BG_EVENT_USB_SMA_EXTIN_ASSERTED | External Input asserted |
| BG_EVENT_USB_SMA_EXTIN_DEASSERTED | External Input de-asserted |
| BG_EVENT_USB_TRIGGER_5GBIT_START | Start 5 Gbit data |
| BG_EVENT_USB_TRIGGER_5GBIT_STOP | Stop 5 Gbit data |

**Table 25** : USB 5000 USB 3.0 LTSSM Event Code definitions

| LTSSM Event Codes for USB 5000 | |
|---|---|
| BG_EVENT_USB_LTSSM_STATE_UNKNOWN | Unknown LTSSM state |
| BG_EVENT_USB_LTSSM_STATE_SS_DISABLED | LTSSM SS.Disabled state |
| BG_EVENT_USB_LTSSM_STATE_SS_INACTIVE | LTSSM SS.Inactive state |
| BG_EVENT_USB_LTSSM_STATE_RX_DETECT_RESET | LTSSM Rx.Detect reset |
| BG_EVENT_USB_LTSSM_STATE_RX_DETECT_ACTIVE | LTSSM Rx.Detect active |
| BG_EVENT_USB_LTSSM_STATE_POLLING_LFPS | LTSSM Polling.LFPS substate |
| BG_EVENT_USB_LTSSM_STATE_POLLING_RXEQ | LTSSM Polling.RxEQ substate |
| BG_EVENT_USB_LTSSM_STATE_POLLING_ACTIVE | LTSSM Polling.Active substate |
| BG_EVENT_USB_LTSSM_STATE_POLLING_CONFIG | LTSSM Polling.Configuration substate |

| | |
|---|---|
| BG_EVENT_USB_LTSSM_STATE_POLLING_IDLE | LTSSM Polling.Idle substate |
| BG_EVENT_USB_LTSSM_STATE_U0 | LTSSM U0 state |
| BG_EVENT_USB_LTSSM_STATE_U1 | LTSSM U1 state |
| BG_EVENT_USB_LTSSM_STATE_U2 | LTSSM U2 state |
| BG_EVENT_USB_LTSSM_STATE_U3 | LTSSM U3 state |
| BG_EVENT_USB_LTSSM_STATE_RECOVERY_ACTIVE | LTSSM Recovery.Active substate |
| BG_EVENT_USB_LTSSM_STATE_RECOVERY_CONFIG | LTSSM Recovery.Configuration substate |
| BG_EVENT_USB_LTSSM_STATE_RECOVERY_IDLE | LTSSM Recovery.Idle substate |
| BG_EVENT_USB_LTSSM_STATE_HOT_RESET_ACTIVE | LTSSM Hot Reset.Active substate |
| BG_EVENT_USB_LTSSM_STATE_HOT_RESET_EXIT | LTSSM Hot Reset.Exit substate |
| BG_EVENT_USB_LTSSM_STATE_LOOPBACK_ACTIVE | LTSSM Loopback.Active substate |
| BG_EVENT_USB_LTSSM_STATE_LOOPBACK_EXIT | LTSSM Loopback.Exit substate |
| BG_EVENT_USB_LTSSM_STATE_COMPLIANCE | LTSSM Compliance Mode state |

### 6.8.2 Using the Beagle USB API

In order to use the USB API with the Beagle USB Protocol Analyzers, a number of subsystems need to be configured properly before capture data can be read. The sequence of typical configuration commands are as follows:

1. Call `bg_open` to open a handle to a Beagle analyzer
2. Call `bg_timeout` to set the read timeout
3. Call `bg_latency` to set the capture latency
4. Call `bg_usb2_capture_buffer_config` or `bg_usb3_capture_buffer_config` to configure the hardware capture buffers.

These functions set the total capture size and how much pre-trigger data will be captured. These functions are supported only for analyzers with on-board triggering capability. Error code of BG_FUNCTION_NOT_AVAILABLE will indicate that the current analyzer does not have this capability.

5. Call `bg_usb_configure` to configure the capture settings. The input arguments determine if a USB 2.0, USB 3.0, or a simultaneous capture is to be performed. This function is also used to set how the capture will be triggered by an event or immediately. Event triggers are supported only for analyzers with on-board triggering capability, and only the Beagle USB 5000 SuperSpeed Protocol Analyzers support USB 3.0 and simultaneous captures. It is not necessary to call this function for either the Beagle 12 or 480, since the only supported settings are default upon opening the analyzer with bg_open .
6. Call one (or more) of the following functions to setup the capture trigger for analyzers with on-board triggering capability:
   - `bg_usb2_simple_match_config` – trigger on GPIO input or USB 2.0 event set in `bg_usb2_digital_out_match`
   - `bg_usb2_complex_match_config_enable` – trigger on USB 2.0 complex match
   - `bg_usb3_simple_match_config` – trigger on simple USB 3.0 match event
   - `bg_usb3_complex_match_config_enable` – trigger on USB 3.0 complex match

   It is not necessary to call any of these functions if the capture has been set to trigger immediately in `bg_usb_configure`.
7. Call `bg_enable` to activate the Beagle analyzer and start monitoring data.

   Once the capture has been triggered, data may be downloaded from the hardware buffer(s) by calling `bg_usb_read` or, for USB 2.0-only captures, `bg_usb2_read`.

### 6.8.3 USB Monitor Interface

#### Check Available Features (bg_usb_features)

```
int bg_usb_features (Beagle beagle);
```

*Return licensed features on analyzer.*

**Arguments**

`beagle` handle of a Beagle analyzer

**Return Value**

This function returns a positive integer that represents what features are licensed on the analyzer, as detailed in Table 26. This function can also return a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

The returned integer is a bitmask that indicates the features which are licensed on the Beagle, as detailed in Table 26.

The Beagle USB 12 and the Beagle USB 480 Protocol Analyzers only return `BG_USB_FEATURE_USB2_MON` since they cannot be licensed for additional features.

$V_{BUS}$ current/voltage monitoring is currently only available for the Beagle USB 480 Power Protocol Analyzer.

**Table 26** : Beagle USB Features

| | |
|---|---|
| `BG_USB_FEATURE_NONE` | No features are licensed |
| `BG_USB_FEATURE_USB2_MON` | USB 2.0 captures may be performed |
| `BG_USB_FEATURE_USB3_MON` | USB 3.0 captures may be performed |
| `BG_USB_FEATURE_SIMUL_23` | Simultaneous USB 2.0 & USB 3.0 captures may be performed |
| `BG_USB_FEATURE_USB3_CMP_TRIG` | USB 3.0 Complex triggering licensed |
| `BG_USB_FEATURE_USB3_4G_MEM` | 4GB buffer memory available |
| `BG_USB_FEATURE_USB2_CMP_TRIG` | USB 2.0 Complex triggering licensed |
| `BG_USB_FEATURE_CROSS_ANALYZER_SYNC` | Analyzers may be synchronized using back panel HDMI ports |

| | |
|---|---|
| BG_USB_FEATURE_IV_MON_LITE | $V_{BUS}$ current/voltage monitoring licensed |

### Read License Key (bg_usb_license_read)

```
int bg_usb_license_read (Beagle   beagle,
                         int      length,
                         u08 *    license);
```

*Read the license key from the devi+ce.*

#### Arguments

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| length | should be BG_USB_LICENSE_LENGTH |
| license | filled with buffer license string |

#### Return Value

This function returns BG_OK or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| BG_FUNCTION_NOT_AVAILABLE | Function not supported by device. |

#### Details

This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

Function will only write license to buffer if length is BG_USB_LICENSE_LENGTH and the license buffer is large enough to accomodate a string of that size.

### Write License Key (bg_usb_license_write)

```
int bg_usb_license_write (Beagle        beagle,
                          int           length,
                          const u08 *   license);
```

*Write the license key to the device.*

#### Arguments

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| length | should be BG_USB_LICENSE_LENGTH |
| license | buffer which contains license string. |

#### Return Value

This function returns BG_OK or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| BG_FUNCTION_NOT_AVAILABLE | Function not supported by device. |
| BG_INVALID_LICENSE | License key was invalid or `length` was not the proper size |

#### Details

This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

Function will only write license string if `length` is BG_USB_LICENSE_LENGTH and the `license` string is valid.

### Configure USB Capture (bg_usb_configure)

```
int bg_usb_configure (Beagle                beagle,
                      u08                   capture_mask,
                      BeagleUsbTriggerMode  trigger_mode);
```

*Configure the capture.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `capture_mask` | bitmask specifying what kind of capture to perform, as shown in Table 27 |
| `trigger_mode` | enumerated value specifying how capture is triggered, as shown in Table 28 |

**Table 27** : Capture Masks

| | |
|---|---|
| BG_USB_CAPTURE_USB2 | Capture USB 2.0 traffic |
| BG_USB_CAPTURE_USB3 | Capture USB 3.0 traffic |
| BG_USB_CAPTURE_IV_MON_LITE | Capture $V_{BUS}$ voltage and current readings |

**Table 28** : `BeagleUsbTriggerMode` enumerated values

| | |
|---|---|
| BG_USB_TRIGGER_MODE_EVENT | Trigger on match event |

| | |
|---|---|
| BG_USB_TRIGGER_MODE_IMMEDIATE | Trigger immediately |

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| BG_CONFIG_ERROR | An attempt was made to configure the Beagle with invalid settings. |
| BG_FUNCTION_NOT_AVAILABLE | An attempt was made to enable an unlicensed or unsupported feature. |

**Details**

Only the Beagle USB 5000 SuperSpeed Protocol Analyzers support USB 3.0 captures.

Only Beagle 5000 units with the option A upgrade are capable of performing simultaneous USB 2.0 and USB 3.0 captures.

Only analyzers with on-board triggering capability support triggering on match event. The default trigger mode is trigger immediate for all Beagle USB abalyzers.

When called with the Beagle USB 12 or the Beagle USB 480 analyzer, the only acceptable values are `cap_mask = BG_USB_CAPTURE_USB2` and `trigger_mode = BG_USB_TRIGGER_MODE_IMMEDIATE` since these devices only have USB 2.0 capture capability and do not have on-board triggering. In fact, it is not necessary to call this function for either the Beagle 12 or 480 since these settings are default upon opening the analyzer with `bg_open`.

BG_USB_CAPTURE_IV_MON_LITE is currently supported by the Beagle USB 480 Power Protocol Analyzer only and it must be specified in conjunction with BG_USB_CAPTURE_USB2. The bitmask will enable the monitoring and delivery of $V_{BUS}$ current and voltage measurements in the USB2 data stream. See `bg_usb_read()` in Section 6.8.3.6 for more details.

### Configure USB Target Power (bg_usb_target_power)

```
int bg_usb_target_power (Beagle                 beagle,
                         BeagleUsbTargetPower   power_flag);
```

*Control $V_{BUS}$ provided to target device.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `power_flag` | enumerated value which controls $V_{BUS}$ as detailed in Table 29 |

**Table 29** : Target Power Enums

| | |
|---|---|
| `BG_USB_TARGET_POWER_HOST_SUPPLIED` | Connect target $V_{BUS}$ to host $V_{BUS}$ |
| `BG_USB_TARGET_POWER_OFF` | Disconnect target $V_{BUS}$ from host $V_{BUS}$ |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

This function can be used to reset the target device by disconnecting $V_{BUS}$.

### Read USB (bg_usb_read)

```
int bg_usb_read (Beagle             beagle,
                 u32 *              status,
                 u32 *              events,
                 u64 *              time_sop,
                 u64 *              time_duration,
                 u32 *              time_dataoffset,
                 BeagleUsbSource *  source,
                 int                max_bytes,
                 u08 *              packet,
                 int                max_k_bytes,
                 u08 *              k_data);
```

*Read data from the analyzer.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `status` | filled with the status bitmask as detailed in Table 11 and Table 19 |

| | |
|---|---|
| `events` | filled with the event bitmask as detailed in Tables 20, 21, 22, 24, 25, and 23 |
| `time_sop` | filled with the timestamp when data appeared on the bus |
| `time_duration` | filled with the number of ticks that it took to read the data |
| `time_dataoffset` | this is always 0 |
| `source` | filled with source of USB packet as detailed in Table 30 |
| `max_bytes` | maximum number of data bytes to read |
| `packet` | an allocated array of `u08` which is filled with the received data |
| `max_k_bytes` | number of k data bytes to read |
| `k_data` | filled with k data flags |

**Table 30** : `BeagleUsbSource` enumerated values

| | |
|---|---|
| `BG_USB_SOURCE_USB3_ASYNC` | Asynchronous stream |
| `BG_USB_SOURCE_USB3_RX` | USB 3.0 Upstream |
| `BG_USB_SOURCE_USB3_TX` | USB 3.0 Downstream |
| `BG_USB_SOURCE_USB2` | USB 2.0 |
| `BG_USB_SOURCE_IV_MON` | $V_{BUS}$ voltage and current stream |

### Return Value

This function returns the number of bytes read or a negative value indicating an error.

### Specific Error Codes

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Beagle is not running a capture or function not supported by device. |
| `BG_CAPTURE_NOT_TRIGGERED` | Capture has not been triggered yet. |

### Details

This function can be called to download data from any Beagle USB analyzer. The argument list is the superset of what is required for USB 2.0 and USB 3.0 traffic. If the analyzer does not support USB 3.0, or is not enabled for USB 3.0 capture, the irrelevant arguments will be unused (and can optionally be set to 0).

The function will block until the requested amount of data is captured, a complete packet with the appropriate end of packet condition is observed, or the bus is idle for longer than the timeout interval set. See Section 6.4.1.12 for information on the `bg_latency()` and `bg_timeout()` functions which affect the behavior of this function.

The `packet` array should be allocated at least as large as `max_bytes`.

All of the timing data is measured in ticks of the sample rate clock. The Beagle USB 12 analyzer is locked to a 48 MHz sample rate, thus each count measures 20.83 ns.

The first byte of the USB packet is the packet ID. An enumeration is provided that defines all the possible packet IDs in Table 18.

In addition to the general read status values in Table 11, there are USB specific status values enumerated in Table 19. The user should be aware of the `BG_READ_USB_END_OF_CAPTURE` status code, which will be returned if the `bg_usb_read()` function is called after a capture has completed.

The `events` enumeration describes specific events that have occurred during the USB capture. By masking the `events` value with the ones detailed in Tables 20, 21, 22, 24, 25, and 23 the user can determine whether a specific event has occurred.

It should also be noted that if a packet is returned when in truncated mode, the packet length will be limited to 4 bytes. The function will still return the true length of the packet, however only up to the first 4 bytes of data will be inserted into the `packet` array. The remaining bytes will be filled with 0s.

Also, the use of digital inputs may cause certain bus events to appear out of order. See Section 3.4.3 for more information.

The `k_data` buffer should be 1/8th the size of `packet`. `k_data` is filled with flags which specify that the corresponding data in `packet` is a D-Symbol or K-Symbol. For example, if bit N of `k_data` is 1, then byte N of `packet` is a K-symbol.

If `source` is `BG_USB_SOURCE_IV_MON` and `events` is non-zero, the packet indicates a $V_{BUS}$ trigger event occurred. If `events` is zero, `packet` contains $V_{BUS}$

voltage and current data. Call `bg_iv_mon_parse()` to parse the packet and retrieve the values. See `bg_iv_mon_parse()` in Section 6.10.2.2 for more details.

### Configure Statistics System (bg_usb_stats_config)

```
int bg_usb_stats_config (Beagle                        beagle,
                         const BeagleUsbStatsConfig *  config);
```

*Configure the hardware-based USB statistics system.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `config` | configuration values for the statistics system |

#### Return Value

This function returns `BG_OK` or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

#### Details

This function is used to configure the hardware-based USB statistics system on the Beagle USB analyzer. This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

The `config` field should be used to pass desired configuration parameters.

```
/* Hardware-based USB statistics configuration */
struct BeagleUsbStatsConfig {
    u08                 auto_config;
    BeagleUsbMatchType  source_match_type;
    BeagleUsbSource     source_match_val;
    BeagleUsbMatchType  ep_match_type;
    u08                 ep_match_val;
    BeagleUsbMatchType  dev_match_type;
    u08                 dev_match_val;
};
```

**Table 31** : `BeagleUsbStatsConfig` field descriptions

| | |
|---|---|
| `auto_config` | A non-zero value enables automatic device address configuration |

| | |
|---|---|
| source_match_type | Enable or disable stream direction matching |
| source_match_val | The source stream (TX/RX) on which to match |
| ep_match_type | Configure or disable endpoint matching |
| ep_match_val | The endpoint number on which to match |
| dev_match_type | Configure or disable device address matching |
| dev_match_val | The device address on which to match |

**Connection-Specific USB 3.0 Statistics**

In addition to tracking USB 3.0 and USB 2.0 bus-level statistics, the hardware-based USB statistics system is capable of tracking statistics that are specific to a particular USB 3.0 device, endpoint and stream direction. This capability should be configured before starting a capture.

Each connection parameter is represented by two separate fields: type and value. The BeagleUsbMatchType enumerated type is used to determine whether a connection value field should be disabled, match on equal, or match on not equal. The different enumerated values are listed below. Restrictions on usage are indicated by footnotes.

- BG_USB_MATCH_TYPE_DISABLED
- BG_USB_MATCH_TYPE_EQUAL
- BG_USB_MATCH_TYPE_NOT_EQUAL (1)

**(1)** Only valid when used in the dev_match_type or ep_match_type fields

The BeagleUsbSource enumerated type is used to choose between the TX and RX stream directions. The different enumerated values are listed below.

- BG_USB_SOURCE_USB3_RX
- BG_USB_SOURCE_USB3_TX

**Automatic Configuration of the Device Address**

To help resolve cases where the device address is unknown, or changes upon each enumeration of the device, an automatic device address configuration feature is available. When this feature is enabled, the analyzer will automatically set the

dev_match_val field to the address observed in the first **Set Address** device request. To enable this feature, pass a non-zero value in the auto_config field.

Auto config affects fields dev_match_type and dev_match_val. While it isnt necessary to configure these two fields to non-zero values when using auto config, endpoint and source stream fields should still be configured. Once the analyzer has determined the device address, the user-provided endpoint and source stream settings will be used to perform statistics tracking.

Automatic device address configuration is not instantaneous, and requires that a **Set Address** device request be observed on the bus by the analyzer. To check the status of device address configuration, use the bg_usb_stats_config_query function, as described below.

When enabled, automatic device address configuration occurs only once until re-enabled by another call to bg_usb_stats_config. A device address set by automatic configuration will persist until analyzer reset, or until another call to this configuration function. In other words, if you use auto configuration at one point, and want to use it again later, say to detect a new device address, you will need to re-enable the feature with a call to bg_usb_stats_config.

### Query Statistics System Configuration (bg_usb_stats_config_query)

```
int bg_usb_stats_config_query (Beagle                  beagle,
                               BeagleUsbStatsConfig *  config);
```

*Query the hardware-based USB statistics system for its current configuration.*

#### Arguments

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| config | filled with the current statistics system configuration |

#### Return Value

This function returns BG_OK or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| BG_FUNCTION_NOT_AVAILABLE | Function not supported by device. |

#### Details

This function is used to query the current configuration of the hardware-based USB statistics system on the Beagle USB analyzer. This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

The `config` field is populated with the current configuration. Most of the fields in the `BeagleUsbStatsConfig` (Table 31) structure will be the same as the values you passed into the `bg_usb_stats_config` function. The main use case of this function is to query the status of **automatic device address configuration**.

Pay close attention to the `auto_config`, `dev_match_type`, and `dev_match_val` fields. If `auto_config` was set to a non-zero value (enabling the automatic configuration feature) on configuration, a value of zero on query indicates successful device address configuration. Once device address configuration is complete, you can check the device address value by inspecting the `dev_match_val` field.

### Reset Statistics Counts (bg_usb_stats_reset)

```
int bg_usb_stats_reset (Beagle beagle);
```

*Reset the hardware-based USB statistics counts.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is used to reset the current hardware-based USB statistics system counts on the Beagle USB analyzer. This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

Statistics counts are summed continuously during a capture. To reset all of the counts, simply call this function with a valid handle.

### Read Statistics Counts (bg_usb_stats_read)

```
int bg_usb_stats_read (Beagle             beagle,
                       BeagleUsbStats *   config);
```

*Read hardware-based USB statistics counts.*

### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `stats` | filled with the hardware-based statistics counts |

### Return Value

This function returns `BG_OK` or a negative value indicating an error.

### Specific Error Codes

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

### Details

This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

This function is used to read hardware-based USB statistics counts from the Beagle USB analyzer. Top-level and children statistics structures are detailed below.

```
/* Hardware-based USB statistics counts */
/* top-level structure                  */
struct BeagleUsbStats {
      BeagleUsb3GenStats    usb3_tx_gen;
      BeagleUsb3GenStats    usb3_rx_gen;
      BeagleUsb3ConnStats   usb3_tx_conn;
      BeagleUsb3ConnStats   usb3_rx_conn;
      BeagleUsb2Stats       usb2;
};
```

**Table 32** : `BeagleUsbStats` field descriptions

| | |
|---|---|
| `usb3_tx_gen` | USB 3.0 TX general counts |
| `usb3_rx_gen` | USB 3.0 RX general counts |
| `usb3_tx_conn` | USB 3.0 TX connection-specific counts |
| `usb3_rx_conn` | USB 3.0 RX connection-specific counts |
| `usb2` | USB 2.0 counts |

**Table 33** : `BeagleUsb3GenStats` field descriptions

| | |
|---|---|
| `link` | Any link commands |
| `lbad` | LBAD link commands |
| `slc_crc5` | CRC5 failing in SLC |

| | |
|---|---|
| txn | Any transaction packets |
| lmp | Any link management packets |
| lgo_ux | LGO_Ux link commands (U1, U2, U3) |
| dp | Any data packets |
| itp | Any isochronous timestamp packets |
| shp_crc16_crc5 | CRC16/CRC5 failing in SHP |
| sdp_crc32 | CRC32 failing in SDP |
| slc_frm_err | SLC framing errors |
| shp_frm_err | SHP framing errors |
| sdp_end_edb_frm_err | SDP/END/EDB framing errors |
| iso_ips | Isolated inter-packet signaling packets |
| para_ips | Parasitic inter-packet signaling packets |
| carry_1k_dp | Number of times data was transferred in chunks of 1 KB |

**Table 34** : BeagleUsb3ConnStats field descriptions

| | |
|---|---|
| txn | Qualified transaction packets |
| dp | Qualified data packets |
| ack | Qualified ACK transaction packets |
| nrdy | Qualified NRDY transaction packets |
| erdy | Qualified ERDY transaction packets |
| retry_ack | Qualified ACK transaction packets with retry flag set |
| carry_1k_dp | Number of times qualified data was transferred in chunks of 1 KB |

**Table 35** : BeagleUsb2Stats field descriptions

| | |
|---|---|
| sof | SOF packets |
| carry_1k_data | Number of times data was transferred in chunks of 1 KB |
| data | Any DATA packets (DATA0/1/2/M) |
| bad_pid | Packets with corrupted PID |
| crc16 | Packets with failing CRC16 |
| crc5 | Packets with failing CRC5 |
| rx_error | Packets marked with rxerror |
| in_nak | IN-NAK packet-pairs |
| ping_nak | PING-NAK packet-pairs |

#### Connection-Specific USB 3.0 Statistics

Connection-specific statistics correspond to a particular USB 3.0 device address, endpoint number and stream direction. Settings for connection-specific statistics must be configured prior to starting a capture, or these statistics will not be tracked.

Use `bg_usb_stats_config` to configure the device, endpoint, and stream direction you would like these statistics fields to track. See Section 6.8.3.7 for details.

#### Resetting Statistics Counts

To manually reset all of the statistics counts, use `bg_usb_stats_reset`. See Section 6.8.3.9 for details.

### 6.8.4 USB Monitor Interface (USB 2.0)

#### Configure USB 2.0 Capture (bg_usb2_capture_config)

```
int bg_usb2_capture_config (Beagle                  beagle,
                            BeagleUsb2CaptureMode   capture_mode);
```

*Configure the capture mode.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `capture_mode` | mode of packet capture as detailed in Table 36 |

**Table 36** : `BeagleUsb2CaptureMode` enumerated values

| | |
|---|---|
| `BG_USB2_CAPTURE_REALTIME` | Configure to real-time capture |
| `BG_USB2_CAPTURE_REALTIME_WITH_PROTECTION` | Configure to real-time capture with overflow protection |
| `BG_USB2_CAPTURE_DELAYED_DOWNLOAD` | Configure to delayed-download mode |

**Return Value**

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

**Specific Error Codes**

| | |
|---|---|
| BG_STILL_ACTIVE | An attempt was made to change the configuration while the capture was still active. |
| BG_CONFIG_ERROR | An attempt was made to set an invalid configuration. |

**Details**

The capture_mode option specifies whether the capture will be in real-time, real-time with truncation, or delayed-download mode. For more details on the different modes of capture, refer to Section 3.4.6.

All Beagle USB analyzers support real-time capture mode, but only the Beagle USB 480 analyzers support real-time with truncation and delayed-download modes.

### Configure Capture (bg_usb2_capture_buffer_config)

```
int bg_usb2_capture_buffer_config (Beagle  beagle,
                                   u32     pretrig_kb,
                                   u32     capture_kb);
```

*Configure USB 2.0 hardware capture buffer.*

**Arguments**

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| pretrig_kb | amount (in kB) of pre-trigger data to capture |
| capture_kb | total amount (in kB) of data to capture |

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| BG_CONFIG_ERROR | An attempt was made to set an invalid configuration. |

**Details**

This function is supported only for analyzers with on-board triggering capability.

The USB 2.0 hardware capture buffer is 128MB, so `capture_kb` may have a maximum value of 131,072.

The size of `capture_kb` includes `pretrig_kb`. Attempting to set `pretrig_kb` greater than `capture_kb` will return an error.

To run an infinite capture, set `capture_kb` to `BG_USB_CAPTURE_SIZE_INFINITE`.

If running a simultaneous capture, it is possible to copy the pretrig-to-capture ratio used for the USB 3.0 capture by assigning `capture_kb` to `BG_USB_CAPTURE_SIZE_SCALE`.

### Query Capture Config (bg_usb2_capture_buffer_config_query)

```
int bg_usb2_capture_buffer_config_query (Beagle   beagle,
                                         u32 *    pretrig_kb,
                                         u32 *    capture_kb);
```

*Query the current USB 2.0 hardware capture buffer configuration.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `pretrig_kb` | filled with USB 2.0 pre-trigger size |
| `capture_kb` | filled with USB 2.0 total capture size |

#### Return Value

This function returns the size of the available USB 2.0 hardware capture buffer.

#### Specific Error Codes

`None.`

#### Details

Query the hardware capture buffer configuration set in `bg_usb2_capture_buffer_config`.

If an infinite or scaled USB 2.0 capture has been configured, `capture_kb` will be filled with the appropriate constant from Table 37.

For analyzers that do not have on-board triggering capability, `pretrig_kb` will return 0, and `capture_kb` will return the buffer size. For the Beagle 480, 65536 (or

64 MB) will be returned for `capture_kb` since there is a 64 MB post-trigger buffer available. For the Beagle 12, `capture_kb` will be returned as 0 since there is essentially no on-board capture buffer.

**Table 37** : `capture_kb` constants

| | |
|---|---|
| `BG_USB_CAPTURE_SIZE_INFINITE` | Infinite capture |
| `BG_USB_CAPTURE_SIZE_SCALE` | Copy pretrig-to-capture ratio from USB 3.0 buffer configuration |

### Configure Target (bg_usb2_target_config)

```
int bg_usb2_target_config (Beagle   beagle,
                           u32      target_config);
```

*Specify the speed of the USB 2.0 target link.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `target_config` | target configuration as detailed in Table 38 which includes the intended speed of the USB 2.0 link and an option to start the capture without the target host $V_{BUS}$ |

**Table 38** : Beagle Usb2 Target Config constants

| | |
|---|---|
| `BG_USB2_AUTO_SPEED_DETECT` | Configure to auto-detect the bus speed |
| `BG_USB2_LOW_SPEED` | Configure to lock to low-speed capture |
| `BG_USB2_FULL_SPEED` | Configure to lock to full-speed capture |
| `BG_USB2_HIGH_SPEED` | Configure to lock to high-speed capture |
| `BG_USB2_VBUS_OVERRIDE` | Configure to not require $V_{BUS}$ or capture |

#### Return Value

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

#### Specific Error Codes

| | |
|---|---|
| BG_STILL_ACTIVE | An attempt was made to change the configuration while the capture was still active. |
| BG_FUNCTION_NOT_AVAILABLE | Function not supported by device. |

**Details**

This function is not supported for the Beagle USB 12 Protocol Analyzers, since the Beagle 12 always performs in auto-detection mode for full- and low-speed buses.

The `target_config` option specifies the speed of communication on the target bus and whether the analyzer should operate without the target host $V_{BUS}$. The Beagle USB Analyzer may be configured to auto-detect the speed, or may alternatively be locked to monitor only a single communication speed. Additionally, the Analyzer also monitors the target host $V_{BUS}$ to detect if a host is present and it will not start capturing traffic if there is no $V_{BUS}$ or if $V_{BUS}$ is under 5 V. This requirement can be overridden by the `BG_USB2_VBUS_OVERRIDE` flag (the speed constants are mutually exclusive but `BG_USB2_VBUS_OVERRIDE` is a bit mask). For more details please refer to Knowledge Base Article 10058.

### Enable Digital Output (bg_usb2_digital_out_config)

```
int bg_usb2_digital_out_config (Beagle   beagle,
                                u08      out_enable_mask,
                                u08      out_polarity_mask);
```

*Enable Beagle analyzer to output a specific match type on output pins (for USB 2.0).*

**Arguments**

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| out_enable_mask | bitmask of enabled output pins as detailed in Table 39 |
| out_polarity_mask | bitmask of polarity on outputs pins as detailed in Table 40 |

**Table 39** : Digital Output Pin Enable bit mask

| | |
|---|---|
| BG_USB2_DIGITAL_OUT_ENABLE_PIN1 | Enables Output Pin 1 |
| BG_USB2_DIGITAL_OUT_ENABLE_PIN2 | Enables Output Pin 2 |
| BG_USB2_DIGITAL_OUT_ENABLE_PIN3 | Enables Output Pin 3 |
| BG_USB2_DIGITAL_OUT_ENABLE_PIN4 | Enables Output Pin 4 |

**Table 40** : Digital Output Pin Polarity bit mask

| | |
|---|---|
| BG_USB2_DIGITAL_OUT_PIN1_ACTIVE_HIGH | Output Pin 1 idles low |
| BG_USB2_DIGITAL_OUT_PIN1_ACTIVE_LOW | Output Pin 1 idles high |
| BG_USB2_DIGITAL_OUT_PIN2_ACTIVE_HIGH | Output Pin 2 idles low |
| BG_USB2_DIGITAL_OUT_PIN2_ACTIVE_LOW | Output Pin 2 idles high |
| BG_USB2_DIGITAL_OUT_PIN3_ACTIVE_HIGH | Output Pin 3 idles low |
| BG_USB2_DIGITAL_OUT_PIN3_ACTIVE_LOW | Output Pin 3 idles high |
| BG_USB2_DIGITAL_OUT_PIN4_ACTIVE_HIGH | Output Pin 4 idles low |
| BG_USB2_DIGITAL_OUT_PIN4_ACTIVE_LOW | Output Pin 4 idles high |

**Return Value**

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

**Specific Error Codes**

| | |
|---|---|
| BG_CONFIG_ERROR | An attempt was made to set an invalid configuration. |
| BG_FUNCTION_NOT_AVAILABLE | Function not supported by device. |

**Details**

This function is not supported for the Beagle USB 12 Protocol Analyzers.

Pins are triggered by particular events which are detailed in Section 3.4.4. Please refer to Section 3.4.4 for the hardware specifications of the output pins.

The `out_enable_mask` input is a bitmask of the parameters listed in Table 39. By using a bit-wise OR operation, multiple output pins can be enabled. It is important to note that calling this function will disable all pins that are not explicitly set in the `out_enable_mask` input.

The `out_polarity_mask` input configures the polarity of the output. Like `out_enable_mask`, this bitmask allows the user to configure multiple pins through a bit-wise OR operation. The default configuration is active low. If a pin is attempted to be configured as both active low and active high, then it will only actually configure to active high.

Digital output lines will activate as soon as their triggering event is fully confirmed.

### Match Digital Output (bg_usb2_digital_out_match)

```
int bg_usb2_digital_out_match (
             Beagle                          beagle,
             BeagleUsb2DigitalOutMatchPins   pin_num,
             BeagleUsb2PacketMatch *         packet_match,
             BeagleUsb2DataMatch *           data_match);
```

*Enable Beagle analyzer to output match on a particular bus data (for USB 2.0).*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `pin_num` | output pins to be enabled as detailed in Table 41 |
| `packet_match` | USB packet header information and Boolean operations that the Beagle analyzer can match packet headers with |
| `data_match` | USB packet data and Boolean operations that the Beagle USB analyzer can match incoming packet data with |

#### Return Value

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

#### Specific Error Codes

| | |
|---|---|
| `BG_STILL_ACTIVE` | An attempt was made to change the configuration while the capture was still active. |
| `BG_CONFIG_ERROR` | An attempt was made to set an invalid configuration. |
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

#### Details

This function is not supported for the Beagle USB 12 Protocol Analyzers.

The function is used to configure the output pins of the digital I/O port to trigger on specific events. This function should be called repeatedly for each pin that must be configured.

Output pins 1 and 2 do not use the `packet_match` and `data_match` inputs, as they do not require that extra information. They are therefore completely

configurable from the `bg_usb2_digital_out_config()` function and calling this function on either of those pins will return `BG_CONFIG_ERROR`.

Output pin 3 does not use the `data_match` input because it does not have that functionality. A dummy structure or null can be used for the `data_match` argument. In either case, the argument is ignored.

The `BeagleUsb2PacketMatch` and `BeagleUsb2DataMatch` must be used to correctly configure the matching capabilities of Output Pins 3 and 4.

The `BeagleUsb2PacketMatch` structure describes the packet parameters that need to be matched.

**Table 41** : `BeagleUsb2DigitalOutMatchPins` enumerated values

| | |
|---|---|
| `BG_USB2_DIGITAL_OUT_MATCH_PIN3` | Selects Output Pin 3 |
| `BG_USB2_DIGITAL_OUT_MATCH_PIN4` | Selects Output Pin 4 |

```
/* Digital ouput matching configuration */
struct BeagleUsb2PacketMatch {
      BeagleUsb2MatchType    pid_match_type;
      u08                    pid_match_val;
      BeagleUsb2MatchType    dev_match_type;
      u08                    dev_match_val;
      BeagleUsb2MatchType    ep_match_type;
      u08                    ep_match_val;
};
```

The `BeagleUsb2DataMatch` structure describes the data sequence that need to be matched.

```
  struct BeagleUsb2DataMatch {
      BeagleUsb2MatchType   data_match_type;
      u08                   data_match_pid;
      u16                   data_length;
      u08 *                 data;
      u16                   data_valid_length;
      u08 *                 data_valid;
  };
```

The `BeagleUsb2MatchType` enumerated type is used throughout the two structures to determine whether the match should assert on the values being equal,

not equal, or dont care (disabled). The different enumerated types are described in the following table.

**Table 42** : `BeagleUsb2MatchType` enumerated values

| | |
|---|---|
| `BG_USB2_MATCH_TYPE_DISABLED` | The match type is disabled |
| `BG_USB2_MATCH_TYPE_EQUAL` | The match type must equal |
| `BG_USB2_MATCH_TYPE_NOT_EQUAL` | The match type must not equal |

The `BeagleUsb2DataMatch` structure has its own field for checking PIDs. This field is a bitmask for each of the four types of data packets and is described in the following table.

**Table 43** : Data Match PID bit mask

| | |
|---|---|
| `BG_USB2_DATA_MATCH_DATA0` | Enable match on data with `DATA0` PID |
| `BG_USB2_DATA_MATCH_DATA1` | Enable match on data with `DATA1` PID |
| `BG_USB2_DATA_MATCH_DATA2` | Enable match on data with `DATA2` PID |
| `BG_USB2_DATA_MATCH_MDATA` | Enable match on data with `MDATA` PID |

Since the `BeagleUsb2DataMatch` has its own fields for matching the PID, using the structure will therefore overwrite the PID settings defined in `BeagleUsb2PacketMatch`. Furthermore, the data matching is determined through two arrays. The `data` array determines which values the user would like to match. The first byte of this array would correlate to the first byte of the packet. The second array, `data_valid`, determines which of those bytes in the `data` array are valid for matching. Setting a byte to zero in the `data_valid` array means that byte is a dont-care condition for the matching algorithm.

The digital outputs activate as soon as their triggering event can be fully confirmed. Thus, Pins 1 and 2 will activate as soon as the capture activates or rxactive goes high, respectively. However, Pins 3 and 4 must assure a match of all of their characteristics. Therefore, only once all possible PIDs, device address, and endpoints of a given packet are checked completely can the output activate. The assertion of matched data on Pin 4 must wait until the end of the data packet to assure a match. Packets that are shorter then what is defined by the `BeagleUsb2DataMatch` structure may still activate Pin 4 if all the data up to that point matched correctly.

### Enable USB 2.0 Digital Input (bg_usb2_digital_in_config)

```
int bg_usb2_digital_in_config (Beagle   beagle,
                               u08      in_enable_mask);
```

*Configures the analyzer to report an event on changes to the external inputs on the Digital I/O port (for USB 2.0).*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `in_enable_mask` | bitmask of enabled input pins as detailed in Table 44 |

**Table 44** : Digital Input Pin Enable bit mask

| | |
|---|---|
| `BG_USB2_DIGITAL_IN_ENABLE_PIN1` | Enable input pin 1 |
| `BG_USB2_DIGITAL_IN_ENABLE_PIN2` | Enable input pin 2 |
| `BG_USB2_DIGITAL_IN_ENABLE_PIN3` | Enable input pin 3 |
| `BG_USB2_DIGITAL_IN_ENABLE_PIN4` | Enable input pin 4 |

**Return Value**

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is not supported for the Beagle USB 12 Protocol Analyzers.

The Beagle USB analyzer digital I/O port has four pins allocated for digital inputs. These digital inputs will display events in-line with collected data. For further details on the digital inputs refer to Section 3.4.4 and Section 3.4.3.

The `in_enable_mask` is a bitmask of the parameters listed in Table 44. By using a bit-wise OR operation, multiple input pins can be enabled. It is important to note that calling this function will disable all pins that are not explicitly set in the `enable_mask` input.

### Enable Simple Matching (bg_usb2_simple_match_config)

```
int bg_usb2_simple_match_config (
     Beagle   beagle,
     u08      dig_in_pin_pos_edge_mask,
     u08      dig_in_pin_neg_edge_mask,
```

```
        u08      dig_out_match_pin_mask);
```

*Configure the USB 2.0 simple matching system for triggering.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `dig_in_pin_pos_edge_mask` | bitmask of positive digital input edge(s) to match on |
| `dig_in_pin_neg_edge_mask` | bitmask of negative digital input edge(s) to match on |
| `dig_out_match_pin_mask` | bitmask of digital output pins to match on |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_CONFIG_ERROR` | An attempt was made to set an invalid configuration. |
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is supported only for analyzers with on-board triggering capability.

Only bits [3:0] can be set in either of the input pin edge masks. Similarly, only bits [3:1] can be set in `dig_out_match_pin_mask`. Setting invalid bits will cause this function to return `BG_CONFIG_ERROR`.

When a simple match is detected, the capture will be triggered. Note that this function is different than `enabled_digital_input`, which reports an event but does not trigger the capture.

### Enable Hardware Filter (bg_usb2_hw_filter_config)

```
int bg_usb2_hw_filter_config (Beagle   beagle,
                              u08      filter_enable_mask};
```

*Specify hardware filtering modes.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

| | |
|---|---|
| `filter_enable_mask` | hardware filtering configuration definitions as detailed in Table 45 |

**Table 45** : Hardware Filter Enable bit mask

| | |
|---|---|
| `BG_USB2_HW_FILTER_PID_SOF` | Filter `SOF` packets |
| `BG_USB2_HW_FILTER_PID_IN` | Filter `IN` + `ACK` `IN` + `NAK` packet groups |
| `BG_USB2_HW_FILTER_PID_PING` | Filter `PING` + `NAK` packet groups |
| `BG_USB2_HW_FILTER_PID_PRE` | Filter `PRE` packet groups |
| `BG_USB2_HW_FILTER_PID_SPLIT` | Filter `SPLIT` packet groups |
| `BG_USB2_HW_FILTER_SELF` | Filter packets intended for Beagle analyzer |

**Return Value**

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is not supported for the Beagle USB 12 Protocol Analyzers.

The Beagle USB Analyzer is capable of filtering out data-less transactions before being saved for capture. This option can be especially useful for saving memory on the analysis PC and on the hardware buffer.

To enable the filtering, simply use the bitmask detailed in Table 45. By using a bit-wise OR operation, multiple filters can be enabled. It is important to note that calling this function will disable all filters that are not explicitly set in the `filter_config` input.

For more detailed information on the hardware filters, please refer to Section 3.4.5.

### Configure External Output (bg_usb2_extout_config)

```
int bg_usb2_extout_config (
Beagle                beagle,
BeagleUsbExtoutType   extout_modulation);
```

*Configure Output Pin 1 settings used for match/action EXTOUT.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `extout_modulation` | mode of EXTOUT signal modulation |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_CONFIG_ERROR` | An attempt was made to set an invalid configuration. |
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is supported only for analyzers with on-board triggering capability.

The EXTOUT modulation specified in this function only applies to Output Pin 1. When the complex match system is enabled, it will override the Output Pin 1 settings as configured in the `usb2_digital_out` functions. Any assertions of the external output by the match/action system are done through Output Pin 1. The modulation on Output Pin 1 in these scenarios is set using this function.

The `BeagleUsbExtoutType` enumerated type is used to set the mode of EXTOUT signal modulation. The different enumerated types are described in the table below. Note that these values are a subset of the values available in USB 3.0 (Section 6.8.5.7).

**Table 46** : `BeagleUsbExtoutType` enumerated values

| | |
|---|---|
| `BG_USB_EXTOUT_POS_PULSE` | Positive pulse |
| `BG_USB_EXTOUT_NEG_PULSE` | Negative pulse |
| `BG_USB_EXTOUT_TOGGLE_0` | Toggle (initial value LOW) |
| `BG_USB_EXTOUT_TOGGLE_1` | Toggle (initial value HIGH) |

### Configure Complex Matching (bg_usb2_complex_match_config)

```
int bg_usb2_complex_match_config (
    Beagle                       beagle,
    u08                          validate,
    u08                          digout,
    BeagleUsb2ComplexMatchState *state_0,
    BeagleUsb2ComplexMatchState *state_1,
    BeagleUsb2ComplexMatchState *state_2,
```

```
    BeagleUsb2ComplexMatchState *state_3,
    BeagleUsb2ComplexMatchState *state_4,
    BeagleUsb2ComplexMatchState *state_5,
    BeagleUsb2ComplexMatchState *state_6,
    BeagleUsb2ComplexMatchState *state_7);
```

*Configure the USB 2.0 complex matching system for triggering.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `validate` | validate a configuration state without actually programming the Beagle analyzer |
| `digout` | enable EXTOUT assertion through digital output pin 1 on complex match |
| `state_N` | data, timer and async match units corresponding to state N (use null value for empty states) |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

* See Section 6.8.5.9. Identical error codes are used.

**Details**

This function is used to configure the USB 2.0 complex matching system. This function is supported only for analyzers with on-board triggering capability and licensed for USB 2.0 Complex triggering.

The `BeagleUsb2ComplexMatchState` fields should be used to form a valid complex matching state machine. The default state, `state_0`, is the entry point of the state machine.

```
 /* Complex match state configuration */
struct BeagleUsb2ComplexMatchState {
      u08                           data_0_valid;
      BeagleUsb2DataMatchUnit       data_0;
      u08                           data_1_valid;
      BeagleUsb2DataMatchUnit       data_1; u
      08                            data_2_valid;
      BeagleUsb2DataMatchUnit       data_2;
      u08                           data_3_valid;
      BeagleUsb2DataMatchUnit       data_3;
      u08                           timer_valid;
      BeagleUsb2TimerMatchUnit      timer;
```

```
    u08 a                              sync_valid;
    BeagleUsb2AsyncEventMatchUnit      async;
    u08                                goto_0;
    u08                                goto_1;
    u08                                goto_2;
};
```

Up to 3 destination states can be defined per state. These states should be set using the `goto_N` fields. Match units can pick one of these 3 destination states using their `goto_selector` (0, 1, or 2) fields.

Each state can accomodate up to 4 data match units, 1 timer match unit, and 1 asynchronous event match unit. To denote a valid match unit object, the corresponding `*_valid` field should contain a non-zero value.

#### Data Match Units

The `BeagleUsb2DataMatchUnit` should be used to create a data match unit. The fields are described in Table 47 and Table 68.

```
/*Data match unit configuration*/
struct BeagleUsb2DataMatchUnit {
    BeagleUsb2PacketType         packet_type;
    BeagleUsb2DataMatchPrefix    prefix;
    u08                          handshake;
    u16                          data_length;
    u08 *                        data;
    u16                          data_valid_length;
    u08 *                        data_valid;
    BeagleUsb2ErrorType          err_match;
    08                           data_properties_valid;
    BeagleUsb2DataProperties     data_properties;
    BeagleUsb2MatchModifier      match_modifier;
    u16                          repeat_count;
    u08                          sticky_action;
    08                           action_mask;
    u08                          goto_selector;
};
```

**Table 47** : `BeagleUsb2DataMatchUnit` field descriptions

| | |
|---|---|
| `packet_type` | The type of packet to be matched |

| | |
|---|---|
| prefix | The PID that should appear before the packet type |
| handshake | Mask of handshakes that should follow the packet type |
| data_length | Length of the data array |
| data | Byte array of data on which to match |
| data_valid_length | Length of the data valid array |
| data_valid | Array specifying which bytes in the data array are valid (non-zero) and which are dont cares (zero) |
| err_match | Dictates which CRC/error conditions must be valid or invalid for a match. Options are listed in Table 51 |
| data_properties_valid | data_properties is valid |
| data_properties | Specific match information for supplied data |
| match_modifier | Modify the matching criteria |

The BeagleUsb2PacketType enumerated type is used in data match units to denote which type of packet is to be matched. The different enumerated types are described in the Table 48.

**Table 48** : BeagleUsb2PacketType enumerated values

| | |
|---|---|
| BG_USB2_MATCH_PACKET_IN | Match on IN packets |
| BG_USB2_MATCH_PACKET_OUT | Match on OUT packets |
| BG_USB2_MATCH_PACKET_SETUP | Match on SETUP packets |
| BG_USB2_MATCH_PACKET_SOF | Match on SOF packets |
| BG_USB2_MATCH_PACKET_DATA0 | Match on DATA0 packets |
| BG_USB2_MATCH_PACKET_DATA1 | Match on DATA1 packets |
| BG_USB2_MATCH_PACKET_DATA2 | Match on DATA2 packets |
| BG_USB2_MATCH_PACKET_MDATA | Match on MDATA packets |
| BG_USB2_MATCH_PACKET_ACK | Match on ACK packets |
| BG_USB2_MATCH_PACKET_NAK | Match on NAK packets |

| | |
|---|---|
| BG_USB2_MATCH_PACKET_STALL | Match on STALL packets |
| BG_USB2_MATCH_PACKET_NYET | Match on NYET packets |
| BG_USB2_MATCH_PACKET_PRE | Match on PRE packets |
| BG_USB2_MATCH_PACKET_ERR | Match on ERR packets |
| BG_USB2_MATCH_PACKET_SPLIT | Match on SPLIT packets |
| BG_USB2_MATCH_PACKET_EXT | Match on EXT packets |
| BG_USB2_MATCH_PACKET_ANY | Match on ANY packets |
| BG_USB2_MATCH_PACKET_DATA0_DATA1 | Match on DATA0 or DATA1 packets |
| BG_USB2_MATCH_PACKET_DATAX | Match on DATA0, DATA1, DATA2, or MDATA packets |
| BG_USB2_MATCH_PACKET_SUBPID_MASK | Mask a standard packet type with this to mark it as a SubPID |
| BG_USB2_MATCH_PACKET_ERROR | Match on various error types |

The `BeagleUsb2DataMatchPrefix` enumerated type is used in data match units to denote what type of PID should precede the desired packet type. This feature is not valid for ERROR types. The different enumerated types are described in the Table 49.

**Table 49** : `BeagleUsb2DataMatchPrefix` enumerated values

| |
|---|
| BG_USB2_MATCH_PREFIX_DISABLED |
| BG_USB2_MATCH_PREFIX_IN |
| BG_USB2_MATCH_PREFIX_OUT |
| BG_USB2_MATCH_PREFIX_SETUP |
| BG_USB2_MATCH_PREFIX_CSPLIT |
| BG_USB2_MATCH_PREFIX_CSPLIT_IN |
| BG_USB2_MATCH_PREFIX_SSPLIT_OUT |
| BG_USB2_MATCH_PREFIX_SSPLIT_SETUP |

The `handshake` parameter takes a bitmask of available packet handshake parameters. This feature is not available for ERROR packet types. The available handshake options are described in Table 50.

**Table 50** : `BeagleUsb2DataMatchUnit handshake` bitmask values

| |
|---|
| BG_USB2_MATCH_HANDSHAKE_MASK_DISABLED |
| BG_USB2_MATCH_HANDSHAKE_MASK_NONE |
| BG_USB2_MATCH_HANDSHAKE_MASK_ACK |
| BG_USB2_MATCH_HANDSHAKE_MASK_NAK |
| BG_USB2_MATCH_HANDSHAKE_MASK_NYET |
| BG_USB2_MATCH_HANDSHAKE_MASK_STALL |

The `BeagleUsb2ErrorType` enumerated type is used differently depending on the match packet type that is configured. In situations where the packet type is not configured to ERROR this field defines the type of CRC condition which is intended to be matched. The various options include looking for failing and passing CRC conditions.

When the match packet type is ERROR, the `BeagleUsb2ErrorType` value is actually used as a bitmask to test for various errors. The possible errors to test for are CRC errors, corrupted PIDs, jabber, and general PHY receive errors. By masking these bits together on an ERROR packet type, multiple error conditions can be matched with a single match unit.

**Table 51** : `BeagleUsb2ErrorType` enumerated values

| When not using ERROR | |
|---|---|
| BG_USB2_MATCH_CRC_DONT_CARE | CRC may be valid or fail |
| BG_USB2_MATCH_CRC_VALID | CRC must be valid |
| BG_USB2_MATCH_CRC_INVALID | CRC must invalid |
| When using ERROR | |
| BG_USB2_MATCH_ERR_MASK_CORRUPTED_PID | Any corrupted PID |
| BG_USB2_MATCH_ERR_MASK_CRC | Any CRC failure |
| BG_USB2_MATCH_ERR_MASK_RXERROR | Any PHY RxError |
| BG_USB2_MATCH_ERR_MASK_JABBER | Any jabber error |

Jabber is matched by a high-speed packet being greater than 1027 bytes, a full-speed packet being greater than 1026 bytes, and a low-speed packet being greater than 11 bytes. These lengths include the PID and CRC.

```
/*Data properties configuration*/
struct BeagleUsb2DataProperties     {
     BeagleUsb2DataMatchDirection   direction;
     BeagleUsbMatchType             ep_match_type;
```

```
    u08                         ep_match_val;
    BeagleUsbMatchType          dev_match_type;
    u08                         dev_match_val;
    BeagleUsbMatchType          data_len_match_type;
    u16                         data_len_match_val;
};
```

The `BeagleUsb2DataMatchDirection` is used to indicate the direction of the USB packet. This is similar in nature to the prefix option but gives a broader sense of direction. For example, the `IN` direction would match all packets that are coming into the host, including `SSPLIT+IN` or `CSPLIT+IN`. The available direction options are described in Table 52.

**Table 52** : `BeagleUsb2DataMatchDirection` enumerated values

| | |
|---|---|
| BG_USB2_MATCH_DIRECTION_DISABLED | Disable direction matching |
| BG_USB2_MATCH_DIRECTION_IN | Match packets going into the host |
| BG_USB2_MATCH_DIRECTION_OUT_SETUP | Match packets going out of the host |
| BG_USB2_MATCH_DIRECTION_SETUP | Match SETUP packets going out of the host |

The `BeagleUsbMatchType` is described in Section 6.8.

The `match_modifier` gives the ability to modify the polarity of the matching conditions. The available match modifiers are described in Table 53. In each case, PID describes the matching of `packet_type`, as well as `prefix` and `handshake`. If any feature is disabled (i.e. the `data_length` or `data_properties_valid` is 0), then that part of the match will always evaluate to true, and then be modified by the `match_modifier`.

**Table 53** : `BeagleUsb2DataMatchModifier` enumerated values

| | |
|---|---|
| BG_USB2_MATCH_MODIFIER_0 | PID & Pattern & Data Property |
| BG_USB2_MATCH_MODIFIER_1 | PID & !(Pattern & Data Property) |
| BG_USB2_MATCH_MODIFIER_2 | !(PID & Pattern & Data Property) |
| BG_USB2_MATCH_MODIFIER_3 | PID & !Pattern & Data Property |

**Resource Limitations**

Due to internal optimizations and constraints, certain resource limitations apply to data match units. The total data memory space (shared across all states) available is 4096 bytes. The maximum allowable length of any data array in a single data match unit is 1024 (bytes).

In addition, some features of the complex match/action system are multiplexed with the simple mode features. These resource limitations affect the following items:

- Output Pin 1
- Output Pin 4 pattern matching

When the complex match system is enabled it will override the Output Pin 1 setting as configured in the `usb2_digital_out` functions. Instead, the configuration as defined by the `bg_usb2_extout_config` and `bg_usb2_complex_match_config` functions are used. Any assertions of the external output by the match/action system are done through Output Pin 1.

When the complex match system is enabled it will disable the ability for Output Pin 4 to match on specific patterns. It will still have the ability to match on PID, device, and endpoints, including the ability to match on various DATA PID types.

If the complex match system is disabled, these features will automatically revert to the latest simple mode configured option. Note that simple mode configurations can be updated even when the complex match system is enabled; they just wont be applied until the complex match system is disabled.

**Timer Match Units**

The `BeagleUsb2TimerMatchUnit` should be used to create a timer match unit.

```
/*Timer match unit configuration*/
struct BeagleUsb2TimerMatchUnit {
     BeagleUsbTimerUnit   timer_unit;
     u32                  timer_val;
     u08                  action_mask;
     u08                  goto_selector;
};
```

The `BeagleUsbTimerUnit` enumerated type is used to define the unit of time for the `timer_val` field. The different enumerated types are described in Table 67.

The validity of `timer_val` depends on the selected units. The timer must be at least 16 ns and at most 71 sec.

#### Asynchronous Event Match Units

The `BeagleUsb2AsyncEventMatchUnit` should be used to create an asynchronous event match unit.

```
/*Async match unit configuration*/
struct BeagleUsb2AsyncEventMatchUnit {
        BeagleUsb2AsyncEventType   event_type;
        u08                        edge_mask;
        u16                        repeat_count;
        u08                        sticky_action;
        u08                        action_mask;
        u08                        goto_selector;
        BeagleUsb2VbusTriggerType  vbus_trigger_type;
        f32                        vbus_trigger_val;
};
```

The `BeagleUsb2AsyncEventType` enumerated type is used to define the event type to match on. The different enumerated types are listed below. Restrictions on `edge_mask` are based on the selected `event_type` and are indicated by footnotes.

- `BG_USB2_COMPLEX_MATCH_EVENT_DIGIN1`
- `BG_USB2_COMPLEX_MATCH_EVENT_DIGIN2`
- `BG_USB2_COMPLEX_MATCH_EVENT_DIGIN3`
- `BG_USB2_COMPLEX_MATCH_EVENT_DIGIN4`
- `BG_USB2_COMPLEX_MATCH_EVENT_CHIRP` (2)
- `BG_USB2_COMPLEX_MATCH_EVENT_SMA_EXTIN`
- `BG_USB2_COMPLEX_MATCH_EVENT_CROSS_TRIGGER` (3)
- `BG_USB2_COMPLEX_MATCH_EVENT_VBUS_TRIGGER`

The `edge_mask` field is a bitmask that specifies the event edge(s) on which to trigger. Zero or more of the following constants may be used.

- `BG_USB_EDGE_RISING`

- BG_USB_EDGE_PULSE
- BG_USB_EDGE_FALLING

The following constants may be used for EVENT_CHIRP.

- BG_USB_EDGE_DEVICE_CHIRP
- BG_USB_EDGE_HOST_CHIRP

**(2)** Bits BG_USB_EDGE_DEVICE_CHIRP and BG_USB_EDGE_HOST_CHIRP are mutually exclusive. Only one of these bits should be set in the event edge_mask

**(3)** Only bit BG_USB_EDGE_PULSE should be set in the event edge_mask

The BeagleUsb2VbusTriggerType enumerated type is used to define the $V_{BUS}$ trigger type to match on. The different enumerated types are listed below. Note that the vbus_trigger_type and vbus_trigger_val fields are valid only when the event_type field is BG_USB2_COMPLEX_MATCH_EVENT_VBUS_TRIGGER.

- BG_USB2_VBUS_TRIGGER_TYPE_CURRENT
- BG_USB2_VBUS_TRIGGER_TYPE_VOLTAGE

vbus_trigger_val is a float value that specifies the threshold. Valid values range from 0 to 24V for BG_USB2_VBUS_TRIGGER_TYPE_VOLTAGE and -3A to 3A for BG_USB2_VBUS_TRIGGER_TYPE_CURRENT.

$V_{BUS}$ trigger is currently avaiable for the Beagle 480 Power Protocol Analyzer, Ultimate Edition only. Capture must be configured with current/voltage monitoring enabled by calling bg_usb_configure() with BG_USB_CAPTURE_USB2|BG_USB_CAPTURE_IV_MON_LITE. Only a single threshold (voltage or current) is supported. BG_COMPLEX_CONFIG_ERROR_NO_MULTI_VBUS_TRIGGERS will be returned if the API is called with multiple states containing different $V_{BUS}$ trigger types or thresholds. For a rising edge trigger ($V_{BUS}$ voltage or current) the specified threshold must be at or above the initial condition. If this is not the case, a multi-state trigger can be used. The first state is to set a falling edge trigger, followed by a rising edge trigger both at the desired threshold.

**Match Actions**

For a description of the actions available for match units, see Table 68 for more details.

### Configure Matching (bg_usb2_complex_match_config_single)

```
int bg_usb2_complex_match_config_single (
    Beagle                          beagle,
    u08                             validate,
    u08                             digout,
    BeagleUsb2ComplexMatchState    *state);
```

*Configure the USB 2.0 complex matching system for triggering.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `validate` | validate a configuration state without actually programming the Beagle analyzer |
| `digout` | enable EXTOUT assertion through digital output pin 1 on complex match |
| `state` | data, timer and async match units corresponding to the initial state |

**Details**

Same as `bg_usb2_complex_match_config`, except that only one state can be provided. This is a convenience function for users that can or want to only use one state. This function will configure state 0, and clear all others.

See Section 6.8.4.11 for more details.

### Enable Complex Matching (bg_usb2_complex_match_enable)

```
int bg_usb2_complex_match_enable (Beagle beagle);
```

*Enable the USB 2.0 complex matching system.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is supported only for analyzers with on-board triggering capability and licensed for USB 2.0 Complex triggering.

Complex matching must first be configured before it can be enabled.

### Disable Complex Matching (bg_usb2_complex_match_disable)

```
int bg_usb2_complex_match_disable (Beagle beagle);
```

*Disable the USB 2.0 complex matching system.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is supported only for analyzers with on-board triggering capability and licensed for USB 2.0 Complex triggering.

Since complex matching only requires a single configuration, complex matching can be re-enabled without reconfiguration after being disabled.

### Query Capture Status (bg_usb2_capture_status)

```
int bg_usb2_capture_status (
    Beagle                beagle,
    BeagleCaptureStatus *  status,
    u32 *                 pretrig_remaining,
    u32 *                 pretrig_total,
    u32 *                 capture_remaining,
    u32 *                 capture_total);
```

*Query the status of USB 2.0 capture.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `status` | filled with enumerated value described in Table 10 |
| `pretrig_remaining` | filled with amount of remaining pre-trigger data to capture (in kB) |
| `pretrig_total` | filled with pre-trigger size set by user (in kB) |
| `capture_remaining` | filled with amount of remaining total capture data to capture (in kB) |
| `capture_total` | filled with total capture size set by user (in kB) |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_USB2_NOT_ENABLED` | A USB 2.0 capture has not been enabled |

**Details**

Query the capture status and the states of the pre-trigger and capture buffers.

Analyzers that do not have the on-board triggering capability will not return status values indicating pre-trigger state. For these analyzers the returned `pretrig_total` and `pretrig_remaining` will always be 0. The Beagle USB 12 and the Beagle USB 480 protocol analyzers will also return 0 for `capture_total`, because neither analyzer has the ability to limit the total capture size.

The `BG_CAPTURE_STATUS_POST_TRIGGER` status indicates that data is being captured post-trigger.

The Beagle USB 480 Protocol Analyzer has an on-board buffer, which when full will cause the analyzer to stop capturing new data while allowing all of the previously captured data to be downloaded. The `BG_CAPTURE_STATUS_TRANSFER` will indicate that the capture has stopped because the buffer became *full*; previous data is still available for download. `capture_remaining` (to be downloaded) will return the amount currently in buffer.

For the Beagle USB 480 analyzer, this function can be useful for delayed-download captures to poll the status of the buffer. However, calling this function issues a short communication between the Beagle USB 480 analyzer and the analysis PC. If the Beagle analyzer is on the same bus that it is monitoring, then calls to this function will take up bus bandwidth and can take up on-board memory space due to the USB

broadcast architecture (see Section 1.1.2.1). If bus bandwidth is a concern, then polling the buffer should be kept to a minimum. If polling is required, then it is recommended that Self Filtering be enabled in order to eliminate the packets intended for the Beagle analyzer, and thus save on-board memory.

For Beagle 12, only the status of `BG_CAPTURE_STATUS_POST_TRIGGER` can be returned, indicating that the capture is running. Because there is no on-board buffer, `capture_remaining` will always be 0.

### Read USB 2.0 (bg_usb2_read)

```
int bg_usb2_read (Beagle    beagle,
                  u32 *     status,
                  u32 *     events,
                  u64 *     time_sop,
                  u64 *     time_duration,
                  u32 *     time_dataoffset,
                  int       max_bytes,
                  u08 *     packet);
```

*Read USB 2.0 data from the USB port.*

#### Arguments

| | |
|---|---|
| `common_args` | see `bg_usb_read()` for common arguments |

#### Return Value

This function returns the number of bytes read or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| `BG_CONFIG_ERROR` | USB 3.0 capture is enabled. |
| `BG_FUNCTION_NOT_AVAILABLE` | Beagle is not running a capture or function not supported by device. |
| `BG_CAPTURE_NOT_TRIGGERED` | Capture has not been triggered yet. |

#### Details

This function is very similar to `bg_usb_read` in Section 6.8.3.6, but it will works only with USB 2.0-only captures. If it is run with a USB 3.0-enabled capture, the function will return an appropriate error code.

### Read USB 2.0 with data-level timing (bg_usb2_read_data_timing)

```
int bg_usb2_read_data_timing (Beagle  beagle,
                              u32 *   status,
                              u32 *   events,
                              u64 *   time_sop,
                              u64 *   time_duration,
                              u32 *   time_dataoffset,
                              int     max_bytes,
                              u08 *   packet,
                              int     max_timing,
                              u32 *   data_timing);
```

*Read USB 2.0 data from the USB port.*

#### Arguments

| | |
|---|---|
| `common_args` | see `bg_usb2_read()` for common arguments |
| `max_timing` | size of `data_timing` array |
| `data_timing` | an allocated array of `u32` which is filled with timing data for each data-word read |

#### Return Value

This function returns the number of bytes read or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

#### Details

This function is supported only for the Beagle USB 12 Protocol Analyzer and is an extension of the `bg_usb2_read()` function with the added feature of byte-level timing. All of the `bg_usb2_read()` arguments and details apply.

The values in the `data_timing` array give the offset of the start of each data word from `time_sop`. For USB, a data word is considered a single byte.

The `data_timing` array should be allocated at least as large as `max_timing`.

### Read USB 2.0 with bit-level timing bg_usb2_read_bit_timing)

```
int bg_usb2_read_bit_timing (Beagle   beagle,
                             u32 *    status,
                             u32 *    events,
                             u64 *    time_sop,
                             u64 *    time_duration,
                             u32 *    time_dataoffset,
                             int      max_bytes,
                             u08 *    packet,
                             int      max_timing,
                             u32 *    bit_timing);
```

*Read USB 2.0 data from the USB port.*

**Arguments**

| | |
|---|---|
| `common_args` | see `bg_usb2_read()` for common arguments |
| `max_timing` | size of `bit_timing` array |
| `bit_timing` | an allocated array of u32 which is filled with the timing data for each bit read |

**Return Value**

This function returns the number of bytes read or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is supported only for the Beagle USB 12 Protocol Analyzer and is an extension of the `bg_usb2_read()` function with the added feature of bit-level timing. All of the `bg_usb2_read()` arguments and details apply.

The values in the `bit_timing` array give the offset of each bit from `time_sop`.

The `bit_timing` array should be allocated at least as large as `max_timing`. Use the function `bg_bit_timing_size()` (in Section 6.4.3.4) to determine how large an array to allocate for `bit_timing`.

### Reconstruct Bit Timing (bg_usb2_reconstruct_timing)

```
int bg_usb2_reconstruct_timing (u32     target_config,
                                int     num_bytes,
                                u08 *   packet,
                                int     max_timing,
                                u32 *   bit_timing);
```

*Reconstruct the bit-level timing of a packet.*

#### Arguments

| | |
|---|---|
| `target_config` | the bus speed of the packet |
| `num_bytes` | number of bytes to do the reconstruction on |
| `packet` | an array containing the packet bytes |
| `max_timing` | maximum number of bits to do the reconstruction on |
| `bit_timing` | allocated array of u32 which is filled with the duration of each of the bits |

#### Return Value

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

#### Specific Error Codes

`None.`

#### Details

All Beagle analyzers except for the Beagle USB 12 analyzer are restricted to packet-level timing of the capture data. However, this function provides a bit-level timing reconstruction based upon the data and the speed of the bus.

The `bit_timing` array will be filled with the duration of each of the bits in the `packet` array. The duration of each bit is provided in counts of a 480 MHz clock, corresponding to approximately a 2 ns resolution. Those bits that are followed by a bit-stuff will have a duration that is twice as long as a normal bit time for that speed.

The `bit_timing` array should be allocated at least as large as `max_timing`. Use the function `bg_bit_timing_size()` (in Section 6.4.3.4) to determine how large an array to allocate for `bit_timing`.

### Read USB 2.0 Statistics Counts (bg_usb2_stats_read)

```
int bg_usb2_stats_read (Beagle            beagle,
                        BeagleUsb2Stats * config);
```

*Read hardware-based USB 2.0 statistics counts.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `stats` | filled with the hardware-based statistics counts |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_CONFIG_ERROR` | USB 2.0 capture is not enabled or USB 3.0 capture is enabled. |
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

This function is very similar to `bg_usb_stats_read` in Section 6.8.3.10, but it will works only with USB 2.0-only captures. If it is run with a USB 3.0-enabled capture, the function will return an appropriate error code.

### Test Memory (bg_usb2_memory_test)

```
int bg_usb2_memory_test (Beagle beagle);
```

*Test the USB 2.0 capture buffer hardware.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

This function returns a memory test result listed in Table 54 or a negative value indicating an error.

**Table 54** : USB Memory Test Results

| | |
|---|---|
| BG_USB_MEMORY_TEST_PASS | Memory test passed |
| BG_USB_MEMORY_TEST_FAIL | Memory test failed |

**Specific Error Codes**

None.

**Details**

This function is not supported for the Beagle USB 12 and the Beagle USB 480 Protocol Analyzers.

This function is used to verify that the USB 2.0 capture buffer hardware is functioning properly. Please contact Total Phase if this function ever returns BG_USB_MEMORY_TEST_FAIL.

### 6.8.5 USB Monitor Interface (USB 3.0)

#### Configure PHY (bg_usb3_phy_config)

```
int bg_usb3_phy_config (Beagle   beagle,
                        u08      tx,
                        u08      rx);
```

*Configure USB 3.0 PHY settings.*

**Arguments**

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| tx | bitmask of PHY settings for Tx channel |
| rx | bitmask of PHY settings for Rx channel |

**Table 55** : BG_USB3_PHY_CONFIG Bitmasks

| | |
|---|---|
| *_POLARITY_NON_INVERT | Force polarity to non-invert |

| | |
|---|---|
| *_POLARITY_INVERT | Force polarity to invert |
| *_POLARITY_AUTO | Auto-detect polarity |
| *_DESCRAMBLER_ON | Force descrambler on |
| *_DESCRAMBLER_OFF | Force descrambler off |
| *_DESCRAMBLER_AUTO | Auto-detect descrambler settings |
| *_RXTERM_ON | Force SuperSpeed Rx termination on |
| *_RXTERM_OFF | Force SuperSpeed Rx termination off |
| *_RXTERM_AUTO | Auto-detect Rx termination |

* – all entries in table are prefixed with BG_USB3_PHY_CONFIG

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| BG_FUNCTION_NOT_AVAILABLE | An attempt was made to call this function with an invalid Beagle product. |

**Details**

This function is used to control the PHY settings of the Beagle USB analyzer. By default, the analyzer will auto-detect the polarity settings, descrambler settings, and SuperSpeed termination of the target host and device.

The PHY settings can be forced into an alternate configuration by calling this function with the proper bitmask. For example, a SuperSpeed device can be forced to operate at high-speed USB by using the following bitmask:

```
BG_USB3_PHY_CONFIG_POLARITY_AUTO     |
BG_USB3_PHY_CONFIG_DESCRAMBLER_AUTO  |
BG_USB3_PHY_CONFIG_RXTERM_OFF
```

### Configure Link (bg_usb3_link_config)

```
int bg_usb3_link_config (
    Beagle                      beagle,
    const BeagleUsb3Channel *   tx,
```

```
    const BeagleUsb3Channel *  rx);
```

*Configure front-end settings of USB 3.0 link.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `tx` | buffer which contains Tx channel configuration |
| `rx` | buffer which contains Rx channel configuration |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `BG_CONFIG_ERROR` | An attempt was made to configure the Beagle with invalid settings. |
| `BG_FUNCTION_NOT_AVAILABLE` | An attempt was made to call this function with an invalid Beagle product. |

**Details**

For the convenience of the user, it is possible to modify the receiver and transmitter settings of the active buffer circuitry.

On the receiver side, users are able to modify the receiver equalization settings, though often this is not necessary.

On the transmitter side, users are able to adjust the signal level of the output. By configuring the levels sent by the transmitter, it is possible to test the sensitivity of the receiver of the USB 3.0 device. The characteristics of the transmitter can also be modified by changing the output pre-emphasis.

Setting for each channel are provided to this function through a `BeagleUsb3Channel` structure, as described below:

```
/* Channel Configuration */
struct BeagleUsb3Channel {
     u08 input_equalization_short;
     u08 input_equalization_medium;
     u08 input_equalization_long;
     u08 pre_emphasis_short_level;
     u08 pre_emphasis_short_decay;
```

```
    u08 pre_emphasis_long_level;
    u08 pre_emphasis_long_decay;
    u08 output_level;
};
```

**Input equalization** can be used to improve the detection of SuperSpeed USB signals which have been degraded from traveling through lossy media. Three time constants (short, medium, and long) may be configured independently using the enumerated values listed in Table 56.

**Table 56** : Input equalization enumerated values

| | |
|---|---|
| BG_USB3_EQUALIZATION_OFF | Filtering off |
| BG_USB3_EQUALIZATION_MIN | Minimum filtering |
| BG_USB3_EQUALIZATION_MOD | Moderate filtering |
| BG_USB3_EQUALIZATION_MAX | Maximum filtering |

**Output pre-emphasis** modifies the SuperSpeed signals output by the Beagle USB analyzer to compensate for the effects of transmission through lossy media. Short pre-emphasis compenstates for transmission through small impedance discontinuities. Long pre-emphasis compenstates for signals which will travel through a long transmission line. Please see below for more details on these struct elements.

- `pre_emphasis_short_level` value may be between 0 and 15, corresponding to a pre-emphasis long level range of 0 dB to 6 dB.
- `pre_emphasis_short_decay` value may be between 0 and 7, corresponding to a pre-emphasis decay range of 500 ps to 1500 ps.
- `pre_emphasis_long_level` value may be between 0 and 15, corresponding to a pre-emphasis long level range of 0 dB to 6 dB.
- `pre_emphasis_long_decay` value may be between 0 and 7, corresponding to a pre-emphasis decay range of 500 ps to 1500 ps.

The **output level** of the SuperSpeed signals can be configured to have a differential peak-to-peak voltage between 405 mV to 990 mV by setting `output_level` to a value between 2 (low) and 13 (high).

The sum of `output_level` and all the `pre_emphasis_*` settings may not exceed 15.

### Configure Capture (bg_usb3_capture_buffer_config)

```
int bg_usb3_capture_buffer_config (Beagle    beagle,
                                   u32       pretrig_kb,
                                   u32       capture_kb);
```

*Configure USB 3.0 hardware capture buffer.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `pretrig_kb` | amount (in kB) of pre-trigger data to capture |
| `capture_kb` | total amount (in kB) of data to capture |

#### Return Value

This function returns `BG_OK` or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| `BG_CONFIG_ERROR` | An attempt was made to set an invalid configuration. |

#### Details

The USB 3.0 hardware buffer is 2 GB for standard units and 4 GB for units with an option B upgrade, so `capture_kb` may have a maximum value of 2,097,152 for standard units or 4,194,304 for option B units.

The size of `capture_kb` includes `pretrig_kb`. Attempting to set `pretrig_kb` greater than `capture_kb` will return an error.

To run an infinite capture, set `capture_kb` to `BG_USB_CAPTURE_SIZE_INFINITE`.

### Query Capture Config (bg_usb3_capture_buffer_config_query)

```
int bg_usb3_capture_buffer_config_query (Beagle    beagle,
                                         u32 *     pretrig_kb,
                                         u32 *     capture_kb);
```

*Query the current USB 3.0 hardware capture buffer configuration.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `pretrig_kb` | filled with USB 3.0 pre-trigger size |
| `capture_kb` | filled with USB 3.0 total capture size |

#### Return Value

This function returns the size of the USB 3.0 memory, as shown in Table 57.

**Table 57** : USB 3.0 memory size constants

| | |
|---|---|
| `BG_USB3_BUFFER_SIZE_2GB` | 2GB buffer size |
| `BG_USB3_BUFFER_SIZE_4GB` | 4GB buffer size |

#### Specific Error Codes

`None.`

#### Details

Very similar to `bg_usb2_capture_buffer_config_query`. See Section 6.8.4.3 for more details.

### Query Capture Status (bg_usb3_capture_status)

```
int bg_usb3_capture_status (
    Beagle                  beagle,
    u32                     timeout_ms,
    BeagleCaptureStatus *   status,
    u32 *                   pretrig_remaining,
    u32 *                   pretrig_total,
    u32 *                   capture_remaining,
    u32 *                   capture_total);
```

*Query the status of USB 3.0 capture.*

#### Arguments

| | |
|---|---|
| `BG_USB3_NOT_ENABLED` | A USB 3.0 capture has not been enabled |

#### Return Value

Input arguments are the same as `bg_usb2_capture_status`. See Section 6.8.4.15 for details.

### Capture Data Truncation (bg_usb3_truncation_mode)

```
int bg_usb3_truncation_mode (
    Beagle    beagle,
    u08       tx_truncation_mode,
    u08       rx_truncation_mode);
```

*Configure the capture truncation mode.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analzyer |
| `tx_truncation_mode` | truncation mode of the host transmission stream (see Table 58 ) |
| `rx_truncation_mode` | truncation mode of the host reception stream (see Table 58 ) |

#### Return Value

This function returns `BG_OK` or a negative value indicating an error.

#### Specific Error Codes

`None.`

#### Details

This function allows users to set a maximum length for the amount of saved data for each packet on a given stream. This truncation is applied before data is written into the hardware memory buffer, and can thus be useful in applications that wish to minimize memory usage on the analyzer and on the analysis computer.

The truncation mode of each stream can be set to 20 symbols, 36 symbols, 68 symbols, or off. These truncation lengths include the packet framing (4 symbols), and thus provide a means for capturing 16 symbols, 32 symbols, or 64 symbols after the packet framing.

The return value of `bg_usb_read()` will still return the true length of the packet on the bus. If a packet is truncated, the read function will set the `BG_USB_TRUNCATION_MODE` flag in the status field. The number of bytes actually available for inspection will be set in the status field as well, and can be found by masking the status with `BG_USB_TRUNCATION_LEN_MASK`.

Even if truncation is enabled, USB 3.0 match units will still be able to test against the full length of the packet.

**Table 58** : USB 3.0 truncation modes

| | |
|---|---|
| BG_USB3_TRUNCATION_OFF | Disable truncation |
| BG_USB3_TRUNCATION_20 | Truncate to 20 symbols |
| BG_USB3_TRUNCATION_36 | Truncate to 36 symbols |
| BG_USB3_TRUNCATION_68 | Truncate to 68 symbols |

### Enable External I/O (bg_usb3_ext_io_config)

```
int bg_usb3_ext_io_config (
    Beagle                beagle,
    bool                  extin_enable,
    BeagleUsbExtoutType   extout_modulation);
```

*Enable the SMA External Inputs / Outputs.*

#### Arguments

| | |
|---|---|
| beagle | handle of a Beagle analyzer |
| extin_enable | a non-zero value enables EXTIN monitoring and matching |
| extout_modulation | mode of EXTOUT signal modulation |

#### Return Value

This function returns BG_OK or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| BG_CONFIG_ERROR | An attempt was made to set an invalid configuration. |

#### Details

The BeagleUsbExtoutType enumerated type is used to set the mode of EXTOUT signal modulation. The different enumerated types are described in the table below.

**Table 59** : BeagleUsbExtoutType enumerated values

| | |
|---|---|
| BG_USB_EXTOUT_LOW | Tie LOW (default) |
| BG_USB_EXTOUT_HIGH | Tie HIGH |
| BG_USB_EXTOUT_POS_PULSE | Positive pulse |
| BG_USB_EXTOUT_NEG_PULSE | Negative pulse |
| BG_USB_EXTOUT_TOGGLE_0 | Toggle (initial value LOW) |
| BG_USB_EXTOUT_TOGGLE_1 | Toggle (initial value HIGH) |

### Enable Simple Matching (bg_usb3_simple_match_config)

```
bg_usb3_simple_match_config (
     Beagle                  beagle,
     u32                     trigger_mask,
     u32                     extout_mask,
     BeagleUsb3ExtoutMode    extout_mode
     u08                     extin_edge_mask,
     BeagleUsb3IPSType       tx_ips_type,
     BeagleUsb3IPSType       rx_ips_type);
```

*Enable the USB 3 simple matching system for triggering.*

#### Arguments

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `trigger_mask` | bitmask of events / sources that will be used for triggering capture |
| `extout_mask` | bitmask of events / sources that will be used for controlling EXTOUT |
| `extout_mode` | mode of EXTOUT assertion |
| `extin_edge_mask` | bitmask of EXTIN edge(s) to match on |
| `tx_ips_type` | mode of downstream IPS matching |
| `rx_ips_type` | mode of upstream IPS matching |

#### Return Value

This function returns `BG_OK` or a negative value indicating an error.

#### Specific Error Codes

| | |
|---|---|
| `BG_CONFIG_ERROR` | An attempt was made to set an invalid configuration. |

#### Details

The `trigger_mask` and `extout_mask` bitmask fields use the same set of constants. Zero or more of the following constants may be used in each of the two fields.

- BG_USB3_SIMPLE_MATCH_NONE
- BG_USB3_SIMPLE_MATCH_SSTX_IPS
- BG_USB3_SIMPLE_MATCH_SSTX_SLC
- BG_USB3_SIMPLE_MATCH_SSTX_SHP

- BG_USB3_SIMPLE_MATCH_SSTX_SDP
- BG_USB3_SIMPLE_MATCH_SSRX_IPS
- BG_USB3_SIMPLE_MATCH_SSRX_SLC
- BG_USB3_SIMPLE_MATCH_SSRX_SHP
- BG_USB3_SIMPLE_MATCH_SSRX_SDP
- BG_USB3_SIMPLE_MATCH_SSTX_SLC_CRC_5A_CRC_5B
- BG_USB3_SIMPLE_MATCH_SSTX_SHP_CRC_5
- BG_USB3_SIMPLE_MATCH_SSTX_SHP_CRC_16
- BG_USB3_SIMPLE_MATCH_SSTX_SDP_CRC
- BG_USB3_SIMPLE_MATCH_SSTX_SLC_SLC_CRC
- BG_USB3_SIMPLE_MATCH_SSRX_SLC_CRC_5A_CRC_5B
- BG_USB3_SIMPLE_MATCH_SSRX_SHP_CRC_5
- BG_USB3_SIMPLE_MATCH_SSRX_SHP_CRC_16
- BG_USB3_SIMPLE_MATCH_SSRX_SDP_CRC
- BG_USB3_SIMPLE_MATCH_SSRX_SLC_SLC_CRC
- BG_USB3_SIMPLE_MATCH_EVENT_SSTX_LFPS
- BG_USB3_SIMPLE_MATCH_EVENT_SSTX_POLARITY
- BG_USB3_SIMPLE_MATCH_EVENT_SSTX_DETECTED
- BG_USB3_SIMPLE_MATCH_EVENT_SSTX_SCRAMBL
- BG_USB3_SIMPLE_MATCH_EVENT_SSRX_LFPS
- BG_USB3_SIMPLE_MATCH_EVENT_SSRX_POLARITY
- BG_USB3_SIMPLE_MATCH_EVENT_SSRX_DETECTED
- BG_USB3_SIMPLE_MATCH_EVENT_SSRX_SCRAMBL
- BG_USB3_SIMPLE_MATCH_EVENT_VBUS_PRESENT
- BG_USB3_SIMPLE_MATCH_EVENT_SSTX_PHYERR
- BG_USB3_SIMPLE_MATCH_EVENT_SSRX_PHYERR
- BG_USB3_SIMPLE_MATCH_EVENT_SMA_EXTIN

The BeagleUsb3ExtoutMode enumerated type is used to set the mode of EXTOUT operation. The different enumerated types are described in Table 60.

**Table 60** : BeagleUsb3ExtoutMode enumerated values

| | |
|---|---|
| BG_USB3_EXTOUT_DISABLED | EXTOUT is disabled |
| BG_USB3_EXTOUT_TRIGGER_MODE | EXTOUT only when capture triggers |
| BG_USB3_EXTOUT_EVENTS_MODE | EXTOUT on every extout_mask source match |

The extin_edge_mask bitmask specifies the EXTIN edge(s) that will cause a match. Zero or more of the follow constants may be used.

- BG_USB_EDGE_RISING
- BG_USB_EDGE_FALLING

The BeagleUsb3IPSType enumerated type is used to specify the IPS type that will cause a match to occur if one of the IPS sources is enabled in trigger_mask or extout_mask. The different enumerated types are described in Table 61.

**Table 61** : BeagleUsb3IPSType enumerated values

| | |
|---|---|
| BG_USB3_IPS_TYPE_DISABLED | IPS matching disabled |
| BG_USB3_IPS_TYPE_TS1 | Match only on TS1 |
| BG_USB3_IPS_TYPE_TS2 | Match only on TS2 |
| BG_USB3_IPS_TYPE_TSEQ | Match only on TSEQ |
| BG_USB3_IPS_TYPE_TSx | Match on TS1 or TS2 |
| BG_USB3_IPS_TYPE_TS_ANY | Match on TS1, TS2 or TSEQ |

### Configure Complex Matching (bg_usb3_complex_match_config)

```
int bg_usb3_complex_match_config (
    Beagle                        beagle,
    u08                           validate,
    u08                           extout,
    BeagleUsb3ComplexMatchState *state_0,
    BeagleUsb3ComplexMatchState *state_1,
    BeagleUsb3ComplexMatchState *state_2,
    BeagleUsb3ComplexMatchState *state_3,
    BeagleUsb3ComplexMatchState *state_4,
```

```
    BeagleUsb3ComplexMatchState *state_5,
    BeagleUsb3ComplexMatchState *state_6,
    BeagleUsb3ComplexMatchState *state_7);
```

*Configure the USB 3.0 complex matching system for triggering.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `validate` | validate a configuration state without actually programming the Beagle analyzer |
| `extout` | enable EXTOUT assertion on complex match |
| `state_N` | data, timer and async match units corresponding to state N (use null value for empty states) |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| `*` | All of the following error codes are prefixed by `BG_COMPLEX_CONFIG_ERROR` (e.g. Eg: `BG_COMPLEX_CONFIG_ERROR_NO_STATES` |
| `*_NO_STATES` | No valid `BeagleUsb3ComplexMatchState` was supplied. |
| `*_DATA_PACKET_TYPE` | A `BeagleUsb3PacketType` value is invalid. |
| `*_DATA_FIELD` | A `data_length` field does not match its `data_valid_length`. |
| `*_ERR_MATCH_FIELD` | The `err_match` field was populated incorrectly. |
| `*_DATA_RESOURCES` | Insufficient data resources to satisfy configuration. |
| `*_DP_MATCH_TYPE` | A `BeagleUsbMatchType` value is invalid. |
| `*_DP_MATCH_VAL` | A `source_match_val` field is invalid. |
| `*_DP_REQUIRED` | A required `BeagleUsb3DataProperties` object was not supplied for a `BG_USB3_MATCH_PACKET_SHP_SDP` `packet_type`. |
| `*_DP_RESOURCES` | Insufficient data properties resources to satisfy configuration. |
| `*_TIMER_UNIT` | A `BeagleUsb3TimerUnit` value is invalid. |
| `*_TIMER_BOUNDS` | A timer value is out of the allowable range. |
| `*_ASYNC_EVENT` | A `BeagleUsb3AsyncEventType` value is invalid. |

| | |
|---|---|
| *_ASYNC_EDGE | A edge_mask field is invalid. |
| *_ACTION_FILTER | A ACTION_FILTER action is being applied to a match unit that is not supported, such as: timer units, asynchronous event units, and ERR packet types. |
| *_ACTION_GOTO_SEL | A goto_selector field is invalid. |
| *_ACTION_GOTO_DEST | A BG_USB_COMPLEX_MATCH_ACTION_GOTO action is being used with an invalid goto_N destination state number. |
| *_BAD_VBUS_TRIGGER_TYPE | A vbus_trigger_type field is invalid. |
| *_BAD_VBUS_TRIGGER_THRES | A vbus_trigger_val field is out of the valid range. |
| *_NO_MULTI_VBUS_TRIGGERS | Only one $V_{BUS}$ trigger threshold is allowed across the states. |
| *_IV_MONITOR_NOT_ENABLED | $V_{BUS}$ trigger is set without enabling current/voltage monitoring. bg_usb_configure() must be called with BG_USB_CAPTURE_USB2\|BG_USB_CAPTURE_IV_MON_LITE prior to this call. |

**Details**

This function is used to configure the USB 3.0 complex matching system on the Beagle USB analyzer.

The BeagleUsb3ComplexMatchState fields should be used to form a valid complex matching state machine. The default state, state_0, is the entry point of the state machine.

```
 /* Complex match state configuration */
struct BeagleUsb3ComplexMatchState {
      u08                              tx_data_0_valid;
      BeagleUsb3DataMatchUnit          tx_data_0;
      u08                              tx_data_1_valid;
      BeagleUsb3DataMatchUnit          tx_data_1;
      u08                              tx_data_2_valid;
      BeagleUsb3DataMatchUnit          tx_data_2;
      u08                              rx_data_0_valid;
      BeagleUsb3DataMatchUnit          rx_data_0;
      u08                              rx_data_1_valid;
      BeagleUsb3DataMatchUnit          rx_data_1;
      u08                              rx_data_2_valid;
      BeagleUsb3DataMatchUnit          rx_data_2;
      u08                              timer_valid;
```

```
      BeagleUsb3TimerMatchUnit       timer;
      u08                            async_valid;
      BeagleUsb3AsyncEventMatchUnit  async;
      u08                            goto_0;
      u08                            goto_1;
      u08                            goto_2;
};
```

Up to 3 destination states can be defined per state. These states should be set using the `goto_N` fields. Match units can pick one of these 3 destination states using their `goto_selector` (0, 1, or 2) fields.

Each state can accomodate up to 3 TX data match units, 3 RX data match units, 1 timer match unit, and 1 asynchronous event match unit. To denote a valid match unit object, the corresponding `*_valid` field should contain a non-zero value.

#### Data Match Units

The `BeagleUsb3DataMatchUnit` should be used to create a data match unit.

```
/*Data match unit configuration*/
struct BeagleUsb3DataMatchUnit {
      BeagleUsb3PacketType          packet_type;
      u16                           data_length;
      u08 *                         data;
      u16                           data_valid_length;
      u08 *                         data_valid;
      BeagleUsb3ErrorType           err_match;
      u08                           data_properties_valid;
      BeagleUsb3DataProperties      data_properties;
      BeagleUsb3MatchModifier       match_modifier;
      u16                           repeat_count;
      u08                           sticky_action;
      u08                           action_mask;
      08                            goto_selector;
};
```

**Table 62** : `BeagleUsb3DataMatchUnit` field descriptions

| | |
|---|---|
| `packet_type` | The type of packet to be matched |
| `data_length` | Length of the data array |
| `data` | Byte array of data on which to match |
| `data_valid_length` | Length of the data valid array |

| | |
|---|---|
| data_valid | Array specifying which bytes in the data array are valid (non-zero) and which are dont cares (zero) |
| err_match | Dictates which CRC/error conditions must be valid or invalid for a match. Options are listed in Table 64 |
| data_properties_valid | data_properties is valid |
| data_properties | Specific match information for supplied data |
| match_modifier | Modify the matching criteria |

The BeagleUsb3PacketType enumerated type is used in data match units to denote which type of packet to be matched. The different enumerated types are described in the Table 63.

**Table 63** : BeagleUsb3PacketType enumerated values

| | |
|---|---|
| BG_USB3_MATCH_PACKET_SLC | Match link command packets |
| BG_USB3_MATCH_PACKET_SHP | Match header packets |
| BG_USB3_MATCH_PACKET_SDP | Match data packets |
| BG_USB3_MATCH_PACKET_SHP_SDP | Match qualified data packets |
| BG_USB3_MATCH_PACKET_TSX | Match TS1 or TS2 |
| BG_USB3_MATCH_PACKET_TSEQ | Match TSEQ |
| BG_USB3_MATCH_PACKET_ERROR | Match on various bit corruptions |
| BG_USB3_MATCH_PACKET_5GBIT_START | Match start of 5 Gb transmission |
| BG_USB3_MATCH_PACKET_5GBIT_STOP | Match exit from 5 Gb transmission |

The BeagleUsb3ErrorType enumerated type is used differently depending on the match packet type that is configured. In situations where the packet type is an SLC, SHP, SDP, or SHP_SDP, this field defines the type of CRC condition which is intended to be matched. The various options include looking for failing and passing CRC conditions in each of the CRC slots. CRC_1 describes the first CRC slot of the packet, while CRC_2 describes the second CRC slot of the packet. Since Data Packet Payloads do not have a second CRC slot, this field is instead used to indicate whether or not the packet ended with an END framing or an EDB framing. A passing CRC_2 describes an END framing, while a failing CRC_2 describes an EDB framing. Please see Table 64 and Table 65 for more information.

When the match packet type is ERROR, the BeagleUsb3ErrorType value is actually used as a bitmask to test for various bit corruptions. The possible errors to test for are CRC errors, framing errors (a single corrupted symbol in the framing), and unknown packets. By masking these bits together on an ERROR packet type, multiple error conditions can be matched with a single match unit.

The BeagleUsb3ErrorType value is not valid for any other packet type.

**Table 64** : BeagleUsb3ErrorType enumerated values

| When using SLC, SHP, SDP, SHP_SDP Packet Types | |
|---|---|
| BG_USB3_MATCH_CRC_DONT_CARE | Both CRCs may be valid or fail |
| BG_USB3_MATCH_CRC_1_VALID | First CRC must be valid |
| BG_USB3_MATCH_CRC_2_VALID | Second CRC must be valid |
| BG_USB3_MATCH_CRC_BOTH_VALID | Both CRCs must be valid |
| BG_USB3_MATCH_CRC_EITHER_FAIL | Either CRC must fail |
| BG_USB3_MATCH_CRC_1_FAIL | First CRC must fail |
| BG_USB3_MATCH_CRC_2_FAIL | Second CRC must fail |
| BG_USB3_MATCH_CRC_BOTH_FAIL | Both CRCs must fail |
| **When using ERR0R Packet Type** | |
| BG_USB3_MATCH_ERR_MASK_CRC | Any CRC failure |
| BG_USB3_MATCH_ERR_MASK_FRAMING | Any framing error |
| BG_USB3_MATCH_ERR_MASK_UNKNOWN | Any unknown packet |

**Table 65** : CRC_1 and CRC_2 Descriptions

| | CRC_1 | CRC_2 |
|---|---|---|
| Link Command | First CRC-5 | Second CRC-5 |
| Header Packet | Header CRC-16 | Link Control Word CRC-5 |
| Data Packet Payload | Data CRC-32 | END/EDB Framing |

```
/*Data properties configuration*/
struct BeagleUsb3DataProperties {
    BeagleUsbMatchType     source_match_type;
    BeagleUsbSource        source_match_val;
    BeagleUsbMatchType     ep_match_type;
    u08                    ep_match_val;
    BeagleUsbMatchType     dev_match_type;
    u08                    dev_match_val;
    BeagleUsbMatchType     stream_id_match_type;
    u16                    stream_id_match_val;
    BeagleUsbMatchType     data_len_match_type;
    u16                    data_len_match_val;
};
```

The BeagleUsbMatchType enumerated type is used in data property objects to determine whether a match field should assert on the values being equal, greater or equal, less or equal, or dont care (disabled). The different enumerated types are listed below. Restrictions on usage are indicated by footnotes.

- BG_USB_MATCH_TYPE_DISABLED
- BG_USB_MATCH_TYPE_EQUAL
- BG_USB_MATCH_TYPE_LESS_EQUAL (4)
- BG_USB_MATCH_TYPE_GREATER_EQUAL (4)

**(4)** Only valid when used in the data_len_match_type field

The BeagleUsbSource enumerated type is used only in the source_match_val field. The enumerated types are listed in Table 30.

Fields ep_match_val, dev_match_val, and data_len_match_val allow any value, subject to unsigned integer range limitations.

The match_modifier gives the ability to modify the polarity of the matching conditions. The available match modifiers are described in Table 66. If any feature is disabled (i.e. the data_length or data_properties_valid is 0), then that part of the match will always evaluate to true, and then be modified by the match_modifier.

**Table 66** : BeagleUsb3DataMatchModifier enumerated values

| | |
|---|---|
| BG_USB3_MATCH_MODIFIER_0 | TYPE & Pattern & Data Property |
| BG_USB3_MATCH_MODIFIER_1 | TYPE & !(Pattern & Data Property) |
| BG_USB3_MATCH_MODIFIER_2 | !(TYPE & Pattern & Data Property) |
| BG_USB3_MATCH_MODIFIER_3 | TYPE & !Pattern & Data Property |

**Resource Limitations**

Due to internal optimizations and constraints, certain resource limitations apply to data match units. The total data memory space (shared across all states) available is 3072 bytes per stream (upstream, downstream). The maximum allowable length of any data array in a single data match unit is 1024 (bytes).

Additional restrictions on data properties objects apply. As a result, when data properties are specified (non-zero `data_properties_valid` ), the total available memory per stream may be less than the 3072 bytes mentioned above.

### Timer Match Units

The `BeagleUsb3TimerMatchUnit` should be used to create a timer match unit.

```
/*Timer match unit configuration*/
struct
     BeagleUsb3TimerMatchUnit {
     BeagleUsbTimerUnit     timer_unit;
     u32                    timer_val;
     u08                    action_mask;
     u08                    goto_selector;
 };
```

The `BeagleUsbTimerUnit` enumerated type is used to define the unit of time for the `timer_val` field. The different enumerated types are described in Table 67.

**Table 67** : `BeagleUsbTimerUnit` enumerated values

| | |
|---|---|
| `BG_USB_TIMER_UNIT_DISABLED` | The timer is disabled |
| `BG_USB_TIMER_UNIT_NS` | The timer value is in ns |
| `BG_USB_TIMER_UNIT_US` | The timer value is in us |
| `BG_USB_TIMER_UNIT_MS` | The timer value is in ms |
| `BG_USB_TIMER_UNIT_SEC` | The timer value is in seconds |

The validity of `timer_val` depends on the selected units. The timer must be at least 8 ns and at most 34.36 sec.

### Asynchronous Event Match Units

The `BeagleUsb3AsyncEventMatchUnit` should be used to create an asynchronous event match unit.

```
/*Timer match unit configuration*/
struct BeagleUsb3AsyncEventMatchUnit {
     BeagleUsb3AsyncEventType   event_type;
     u08                        edge_mask;
     u16                        repeat_count;
```

```
        u08                         sticky_action;
        u08                         action_mask;
        u08                         goto_selector;
};
```

The BeagleUsb3AsyncEventType enumerated type is used to define the event type to match on. The different enumerated types are listed below. Restrictions on edge_mask are based on the selected event_type and are indicated by footnotes.

- BG_USB3_COMPLEX_MATCH_EVENT_SSTX_LFPS
- BG_USB3_COMPLEX_MATCH_EVENT_SSTX_POLARITY (5)
- BG_USB3_COMPLEX_MATCH_EVENT_SSTX_DETECTED
- BG_USB3_COMPLEX_MATCH_EVENT_SSTX_SCRAMBL (5)
- BG_USB3_COMPLEX_MATCH_EVENT_SSRX_LFPS
- BG_USB3_COMPLEX_MATCH_EVENT_SSRX_POLARITY (5)
- BG_USB3_COMPLEX_MATCH_EVENT_SSRX_DETECTED
- BG_USB3_COMPLEX_MATCH_EVENT_SSRX_SCRAMBL (5)
- BG_USB3_COMPLEX_MATCH_EVENT_VBUS_INRUSH
- BG_USB3_COMPLEX_MATCH_EVENT_VBUS_PRESENT
- BG_USB3_COMPLEX_MATCH_EVENT_SSTX_PHYERR (6)
- BG_USB3_COMPLEX_MATCH_EVENT_SSRX_PHYERR (6)
- BG_USB3_COMPLEX_MATCH_EVENT_SMA_EXTIN

The edge_mask field is a bitmask that specifies the event edge on which to trigger. Zero or more of the following constants may be used.

- BG_USB_EDGE_RISING
- BG_USB_EDGE_PULSE
- BG_USB_EDGE_FALLING

**(5)** For the sake of clarity, bits BG_USB_EDGE_RISING and BG_USB_EDGE_FALLING must both be set in the event edge_mask, as the analyzer will always match on either edge for these event types

**(6)** Only bit BG_USB_EDGE_PULSE should be set in the event edge_mask

**Match Actions**

Each match unit can perform certain actions on a match event.

**Table 68** : Match unit action field descriptions

| | |
|---|---|
| repeat_count | The number of **repeated** matches (after the first match) that must occur before a match action is executed |
| sticky_action | A non-zero value will cause actions (other than goto) to be repeatedly executed while the number of repeated actions is less than or equal to repeat_count |
| action_mask | A bitmask of the action(s) that will be taken if a match situation occurs |
| goto_selector | The goto_N field from the complex match state object (not the destination state) to use for the goto action (Valid values: 0, 1, 2) |

The action_mask field can contain zero or more of the following actions:

- BG_USB_COMPLEX_TRIGGER_ACTION_EXTOUT
- BG_USB_COMPLEX_TRIGGER_ACTION_TRIGGER
- BG_USB_COMPLEX_TRIGGER_ACTION_FILTER (7)
- BG_USB_COMPLEX_TRIGGER_ACTION_GOTO

**(7)** Only valid in packet-based data match units (not in timer or asynchronous event match units)

### Configure Matching (bg_usb3_complex_match_config_single)

```
int bg_usb3_complex_match_config_single (
      Beagle                          beagle,
      u08                             validate,
      u08                             extout,
      BeagleUsb3ComplexMatchState    *state);
```

*Configure the USB 3.0 complex matching system for triggering.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `validate` | validate a configuration state without actually programming the Beagle analyzer |
| `extout` | enable EXTOUT assertion on complex match |
| `state` | data, timer and async match units corresponding to the initial state |

**Specific Error Codes**

| | |
|---|---|
| `BG_FUNCTION_NOT_AVAILABLE` | Function not supported by device. |

**Details**

Same as `bg_usb3_complex_match_config`, except that only one state can be provided. This is a convenience function for users that can or want to only use one state. This function will configure state 0, and clear all others.

See Section 6.8.5.9 for more details.

### Enable Complex Matching (bg_usb3_complex_match_enable)

```
int bg_usb3_complex_match_enable (Beagle beagle);
```

*Enable the USB 3.0 complex matching system.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

Complex matching must first be configured in `bg_usb3_complex_match_config` before it can be enabled.

### Disable Complex Matching (bg_usb3_complex_match_disable)

```
int bg_usb3_complex_match_disable (Beagle beagle);
```

*Disable the USB 3.0 complex matching system.*

**Arguments**

`beagle` handle of a Beagle analyzer

**Return Value**

This function returns `BG_OK` or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

Since complex matching only requires a single configuration, complex matching can be re-enabled without reconfiguration after being disabled.

### Test Memory (bg_usb3_memory_test)

```
int bg_usb3_memory_test (
    Beagle                      beagle,
    BeagleUsb3MemoryTestType    test);
```

*Test the USB 3.0 capture buffer hardware.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `test` | an enumerated value which determines what kind of memory test is run, as specified in Table 69 |

**Table 69** : `BeagleUsbMemoryTestType` enumerated values

| | |
|---|---|
| `BG_USB_MEMORY_TEST_FAST` | Fast memory test |
| `BG_USB_MEMORY_TEST_FULL` | Thorough memory test |
| `BG_USB_MEMORY_TEST_SKIP` | Memory test not performed |

**Return Value**

This function returns a memory test result listed in Table 54 or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

This function is used to verify that the USB 3.0 circular buffer hardware is functioning properly. Please contact Total Phase if this function ever returns `BG_USB_MEMORY_TEST_FAIL`.

### 6.8.6 USB 5000 Monitor Interface

#### Configure USB 5000 Cross-Analyzer Sync (bg5000_cross_analyzer_sync_configure)

```
int bg5000_cross_analyzer_sync_config (
    Beagle                          beagle,
    Beagle5000CrossAnalyzerSyncMode cross_sync_mode,
    Beaglb5000CrossAnalyzerMode     cross_trigger_mode,
    Beagsb5000CrossAnalyzerMode     cross_stop_mode);
```

*Configure Cross-Analyzer Sync.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `cross_sync_mode` | enumerated value (Table 70) that can enable or disable the Cross-Analyzer Sync feature |
| `cross_trigger_mode` | enumerated value (Table 71 ) that configures the Cross-Analyzer Trigger feature |
| `cross_stop_mode` | enumerated value (Table 71) that configures the Cross-Analyzer Stop feature |

**Table 70** : `Beagle5000CrossAnalyzerSyncMode` enumerated values

| | |
|---|---|
| `BG5000_CROSS_ANALYZER_SYNC_WAIT` | Wait for all analyzers connected by Cross-Analyzer Sync to start before proceeding with capture. |
| `BG5000_CROSS_ANALYZER_SYNC_BYPASS` | Bypass the Cross-Analyzer Sync feature, proceeding with capture immediately after start. |

**Table 71** : `Beagle5000CrossAnalyzerMode` enumerated values

| | |
|---|---|
| BG5000_CROSS_ANALYZER_ACCEPT | Allow / honor Cross-Analyzer signal from connected analyzers. |
| BG5000_CROSS_ANALYZER_IGNORE | Reject / ignore Cross-Analyzer signal from connected analyzers. |

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| BG_CONFIG_ERROR | An attempt was made to configure the Beagle with invalid settings. |
| BG_FUNCTION_NOT_AVAILABLE | The Beagle analyzer being configured is not licensed to use the Cross-Analyzer Sync feature. |

**Details**

This function can only be used with a Beagle 5000 v2.00 or later analyzer licensed to use Cross-Analyzer Sync.

When BG5000_CROSS_ANALYZER_SYNC_BYPASS is used to completely bypass the Cross-Analyzer Sync feature, the analyzer will no longer accept any Cross-Analyzer trigger or stop signals, nor will the analyzer output trigger or stop signals to other analyzers.

Using BG5000_CROSS_ANALYZER_IGNORE to ignore a Cross-Analyzer trigger or stop signal will not affect the output of that signal by the analyzer. If the analyzer is participating in Cross-Analyzer Sync, the trigger and stop signals will be outputted, even if one or both of those signals is being ignored on input.

### Release from USB 5000 Cross-Analyzer Sync (bg5000_cross_analyzer_sync_release)

```
int bg5000_cross_analyzer_sync_release (Beagle beagle);
```

*Release the analyzer from Cross-Analyzer Sync for the remainder of the current capture.*

**Arguments**

| | |
|---|---|
| beagle | handle of a Beagle analyzer |

**Return Value**

This function returns BG_OK or a negative value indicating an error.

**Specific Error Codes**

| | |
|---|---|
| BG_FUNCTION_NOT_AVAILABLE | The Beagle analyzer being configured is not licensed to use the Cross-Analyzer Sync feature. |
| BG_USB_NOT_ENABLED | Capture has not been enabled. |

**Details**

This function can only be used with a Beagle 5000 v2.00 or later analyzer licensed to use Cross-Analyzer Sync.

This function releases the analyzer from Cross-Analyzer Sync in the same way as function `bg5000_cross_analyzer_sync_config` called with `BG5000_CROSS_ANALYZER_ SYNC_BYPASS`. The latter function is only available before capture starts, while this function is only available during capture.

Unlike `bg5000_cross_analyzer_sync_config`, changes to Cross-Analyzer Sync made by this function do not persist across captures only the current capture is affected.

When an analyzer is released from Cross-Analyzer Sync, the analyzer will no longer accept any Cross-Analyzer trigger or stop signals, nor will the analyzer output trigger or stop signals to other analyzers.

## 6.9 MDIO API

### 6.9.1 Notes

The MDIO API functions are only available for the Beagle $I^2C$/SPI/MDIO Protocol Analyzer.

### 6.9.2 MDIO Monitor Interface

**Read MDIO (bg_mdio_read)**

```
int bg_mdio_read (Beagle  beagle,
                  u32 *   status,
                  u64 *   time_sop,
                  u64 *   time_duration,
                  u32 *   time_dataoffset,
                  u32 *   data_in);
```

*Read data from the MDIO port.*

**Arguments**

| | |
|---|---|
| `beagle` | handle of a Beagle analyzer |
| `status` | filled with the status bitmask as detailed in Table 11 |
| `time_sop` | filled with the timestamp when the frame preamble begins |
| `time_duration` | filled with the number of ticks that from `time_sop` to the last bit of the MDIO frame |
| `time_dataoffset` | filled with the number of ticks from `time_sop` until the end of the preamble |
| `data_in` | a pointer to a `u32` value which is filled with the received MDIO data |

**Return Value**

This function returns the number of bytes read or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

The function will block until a complete frame is captured or the bus is idle for longer than the timeout interval set. See Section 6.4.1.12 for information on the `bg_latency()` and `bg_timeout()` functions which affect the behavior of this function.

All of the timing data is measured in ticks of the sample clock.

### Read MDIO with bit-level timing (bg_mdio_read_bit_timing)

```
int bg_mdio_read_bit_timing (Beagle  beagle,
                             u32 *   status,
                             u64 *   time_sop,
                             u64 *   time_duration,
                             u32 *   time_dataoffset,
                             u32 *   data_in int max_timing,
                             u32 *   bit_timing);
```

*Read data from the MDIO port.*

**Arguments**

| | |
|---|---|
| `common_args` | see `bg_mdio_read()` for common arguments |
| `max_timing` | size of `bit_timing` array |
| `bit_timing` | an allocated array of `u32` which is filled with the timing data for each bit read |

**Return Value**

This function returns the number of bytes read or a negative value indicating an error.

**Specific Error Codes**

`None.`

**Details**

This function is an extension of the `bg_mdio_read()` function with the added feature of bit-level timing. All of the `bg_mdio_read()` arguments and details apply.

The values in the `bit_timing` array give the offset of each bit from `time_sop`.

The `bit_timing` array should be allocated at least as large as `max_timing`. Use the function `bg_bit_timing_size()` (in Section 6.4.3.4) to determine how large an array to allocate for `bit_timing`.

The bit time for the final bit of the frame is always zero. This is due to the fact that the bit times are measured between rising edges of the MDC line. The first bit time is measured from the first rising edge of the MDC line to the next rising edge. For the last bit of a frame, there may not be a subsequent rising edge of the MDC line until the next frame. Therefore, no bit time value can be determined for final bit of a frame.

### Parse MDIO data (bg_mdio_parse)

```
int bg_mdio_parse (u32    packet,
                   u08 *  clause,
                   u08 *  opcode,
                   u08 *  addr1,
                   u08 *  addr2,
                   u16 *  data);
```

*Parses packet into field values.*

**Arguments**

| | |
|---|---|
| `packet` | the MDIO frame to parse |

| | |
|---|---|
| clause | filled with the clause of the frame as detailed in Table 72 |
| opcode | filled with the OP code of the frame as detailed in Table 73 |
| addr1 | filled with the value of the first address field (PHY in Clause 22, port in Clause 45) |
| addr2 | filled with the value of the second address field (reg in Clause 22, device in Clause 45) |
| data | filled with the contents of the data portion of the frame |

**Table 72** : MDIO Clause definitions

| | | |
|---|---|---|
| BG_MDIO_CLAUSE_22 | 0x00 | MDIO Clause 22 |
| BG_MDIO_CLAUSE_45 | 0x01 | MDIO Clause 45 |
| BG_MDIO_CLAUSE_ERROR | 0x02 | Unknown value in clause field |

**Table 73** : MDIO Opcode definitions

| | | |
|---|---|---|
| BG_MDIO_OPCODE22_WRITE | 0x01 | Clause 22 write OP code |
| BG_MDIO_OPCODE22_READ | 0x02 | Clause 22 read OP code |
| BG_MDIO_OPCODE22_ERROR | 0xff | Clause 22 unknown OP code |
| BG_MDIO_OPCODE45_ADDR | 0x00 | Clause 45 address OP code |
| BG_MDIO_OPCODE45_WRITE | 0x01 | Clause 45 write OP code |
| BG_MDIO_OPCODE45_READ_POSTINC | 0x02 | Clause 45 post read increment address OP code |
| BG_MDIO_OPCODE45_READ | 0x03 | Clause 45 read OP code |

### Return Value

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

### Specific Error Codes

| | |
|---|---|
| BG_MDIO_BAD_TURNAROUND | An unexpected value in turnaround field of the frame. |

### Details

The return value will indicate validity of the turnaround field. BG_OK indicates the value of the turnaround field is valid. BG_MDIO_BAD_TURNAROUND indicates an invalid value in the turnaround field.

# 6.10 Current/Voltage Monitoring API

## 6.10.1 Notes

The Current/Voltage Monitoring API is currently only available for the Beagle USB 480 Power Protocol Analyzer.

## 6.10.2 Current/Voltage Monitoring Interface

### Read Current/Voltage Monitoring data (bg_usb_read)

Current/Voltage Monitoring data are delivered in the same data stream as USB records. The capture must be first configured by calling `bg_usb_configure()` with `BG_USB_CAPTURE_IV_MON_LITE` and `BG_USB_CAPTURE_USB2`. Packets of `BeagleUSBSource` = `BG_USB_SOURCE_IV_MON` can then be passed to `bg_iv_mon_parse()` to be parsed into voltage and current values. There is no separate API for reading current/voltage monitoring data only. See section 6.8.3.6 for the interface of `bg_usb_read()`.

### Parse Current/Voltage Monitoring data (bg_iv_mon_parse)

```
int bg_iv_mon_parse (int    length,
                     u08 *  packet,
                     f32 *  voltage,
                     f32 *  current);
```

*Parses packet into voltage and current values.*

**Arguments**

| | |
|---|---|
| `packet` | from `bg_usb_read()` when `BeagleUSBSource` is `BG_USB_SOURCE_IV_MON` |
| `length` | from `bg_usb_read()` when `BeagleUSBSource` is `BG_USB_SOURCE_IV_MON` |
| `voltage` | filled with the voltage value captured in the packet |
| `current` | filled with the current value captured in the packet |

**Return Value**

A Beagle status code of BG_OK is returned on success or an error code as detailed in Table 74.

**Specific Error Codes**

| | |
|---|---|
| BG_IV_MON_NULL_PACKET | `packet` is NULL or `length` is 0. |
| BG_IV_MON_INVALID_PAKCET_LENGTH | `length` too small to be a valid current/voltage monitor packet. |

**Details**

FVoltage is returned in units of Volts (V) and current in Amperes (A).

## 6.11 Error Codes

**Table 74** : Beagle API Error Codes

| Literal Name | Value | bg_status_string() return value |
|---|---|---|
| BG_OK | 0 | ok |
| BG_UNABLE_TO_LOAD_LIBRARY | -1 | unable to load library |
| BG_UNABLE_TO_LOAD_DRIVER | -2 | unable to load usb driver |
| BG_UNABLE_TO_LOAD_FUNCTION | -3 | unable to load function |
| BG_INCOMPATIBLE_LIBRARY | -4 | incompatible library version |
| BG_INCOMPATIBLE_DEVICE | -5 | incompatible device version |
| BG_INCOMPATIBLE_DRIVER | -6 | incompatible driver version |
| BG_COMMUNICATION_ERROR | -7 | communication error |
| BG_UNABLE_TO_OPEN | -8 | unable to open device |
| BG_UNABLE_TO_CLOSE | -9 | unable to close device |
| BG_INVALID_HANDLE | -10 | invalid device handle |
| BG_CONFIG_ERROR | -11 | configuration error |
| BG_UNKNOWN_PROTOCOL | -12 | unknown beagle protocol |
| BG_STILL_ACTIVE | -13 | beagle still active |
| BG_FUNCTION_NOT_AVAILABLE | -14 | beagle function not available |
| BG_CAPTURE_NOT_TRIGGERED | -15 | capture not yet triggered |
| BG_INVALID_LICENSE | -16 | invalid license detected |
| BG_COMMTEST_NOT_AVAILABLE | -100 | comm test feature not available |
| BG_COMMTEST_NOT_ENABLED | -101 | comm test not enabled |

| | | |
|---|---|---|
| BG_I2C_NOT_AVAILABLE | -200 | i2c feature not available |
| BG_I2C_NOT_ENABLED | -201 | i2c not enabled |
| BG_SPI_NOT_AVAILABLE | -300 | spi feature not available |
| BG_SPI_NOT_ENABLED | -301 | spi not enabled |
| BG_USB_NOT_AVAILABLE | -400 | usb feature not available |
| BG_USB_NOT_ENABLED | -401 | usb not enabled |
| BG_USB2_NOT_ENABLED | -402 | usb 2.0 capture no enabled |
| BG_USB3_NOT_ENABLED | -403 | USB 3.0 capture no enabled |
| `BG_CROSS_ANALYZER_SYNC_DISTURBED_RE_ENABLE` | -410 | cross-analyzer sync disturbed, re-enable capture |
| `BG_CROSS_ANALYZER_SYNC_DISTURBED_RECONNECT` | -411 | cross-analyzer sync disturbed, reconnect to analyzer |
| BG_MDIO_NOT_AVAILABLE | -500 | mdio feature not available |
| BG_MDIO_NOT_ENABLED | -501 | mdio not enabled |
| BG_MDIO_BAD_TURNAROUND | -502 | mdio bad turnaround field |
| BG_IV_MON_NULL_PACKET | -600 | null packet or length to `bg_iv_mon_parse()` |
| BG_IV_MON_INVALID_PACKET_LENGTH | -601 | packet length to `bg_iv_mon_parse()` too small |

# 7 Legal / Contact

## 7.1 Disclaimer

All of the software and documentation provided in this datasheet, is copyright Total Phase, Inc. ("Total Phase"). License is granted to the user to freely use and distribute the software and documentation in complete and unaltered form, provided that the purpose is to use or evaluate Total Phase products. Distribution rights do not include public posting or mirroring on Internet websites. Only a link to the Total Phase download area can be provided on such public websites.

Total Phase shall in no event be liable to any party for direct, indirect, special, general, incidental, or consequential damages arising from the use of its site, the software or documentation downloaded from its site, or any derivative works thereof, even if Total Phase or distributors have been advised of the possibility of such damage. The software, its documentation, and any derivative works is provided on an "as-is" basis, and thus comes with absolutely no warranty, either express or implied. This disclaimer includes, but is not limited to, implied warranties of merchantability, fitness for any particular purpose, and non-infringement. Total Phase and distributors have no obligation to provide maintenance, support, or updates.

Information in this document is subject to change without notice and should not be construed as a commitment by Total Phase. While the information contained herein is believed to be accurate, Total Phase assumes no responsibility for any errors and/or omissions that may appear in this document.

## 7.2 Life Support Equipment Policy

Total Phase products are not authorized for use in life support devices or systems. Life support devices or systems include, but are not limited to, surgical implants, medical systems, and other safety-critical systems in which failure of a Total Phase product could cause personal injury or loss of life. Should a Total Phase product be used in such an unauthorized manner, Buyer agrees to indemnify and hold harmless Total Phase, its officers, employees, affiliates, and distributors from any and all claims arising from such use, even if such claim alleges that Total Phase was negligent in the design or manufacture of its product.

## 7.3 Contact Information

Total Phase can be found on the Internet at http://www.totalphase.com/. If you have support-related questions, please email the product engineers at support@totalphase.com. For sales inquiries, please contact sales@totalphase.com.

©2005-2013 Total Phase, Inc. All rights reserved. The Total Phase name and logo and all product names and logos are trademarks of Total Phase, Inc. All other trademarks and service marks are the property of their respective owners.