Beagle USB Protocol Analyzer Features

- Non-intrusive Full-speed USB Monitoring (12 Mbps)
- Non-intrusive Low-speed USB Monitoring (1.5 Mbps)
- Monitor packets in real-time as they appear on the bus.
- Repetitive packet compression
- Bit-level timing with 21 ns resolution.
- High-Speed USB Device (480 Mbps transfer to host PC)
- Linux and Windows compatible
- Low cost

Beagle I^{2}C/SPI Protocol Analyzer Features

- Non-intrusive I^{2}C monitoring up to 4 MHz
- Non-intrusive SPI monitoring up to 24 MHz
- Monitor packets in real-time as they appear on the bus.
- User selectable bit-level timing (up to 20 ns resolution).
- High-Speed USB Device (480 Mbps transfer to host PC)
- Linux and Windows compatible
- Low cost

Summary

The Beagle™ Protocol Analyzers are non-intrusive debugging tools. Developers can watch data in real-time as they occur. The data is appropriately parsed for the protocol of interest. Like all Total Phase products, the Beagle analyzer is a cross-platform device for Windows and Linux.

TOTAL PHASE

BEAGLE™ PROTOCOL ANALYZER

Beagle
Protocol Analyzers

Data Sheet v1.11
August 31, 2006

1 General Overview

1.1 USB Background

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 bus 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 Implementer's 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.

The USB standard specifies several different flavors of USB: Low-Speed, Full-Speed and High-Speed. 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.

The key difference between Low, Full, and High speed is bandwidth.

Low	1.5 Mbps
Full	12 Mbps
High	480 Mbps

The USB specification can be viewed and downloaded on the USB-IF website.

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 port. A host PC can have multiple ports which increases the maximum number of USB devices that can be connected to a single computer.

These 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 one 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 digital camera may present several devices to the host when it is connected via USB. It can present a Camera Device, a Mass Storage Device, etc.

All the devices on a single USB bus must share the bandwidth that is available on the bus. It is possible for a host PC to have multiple busses 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.

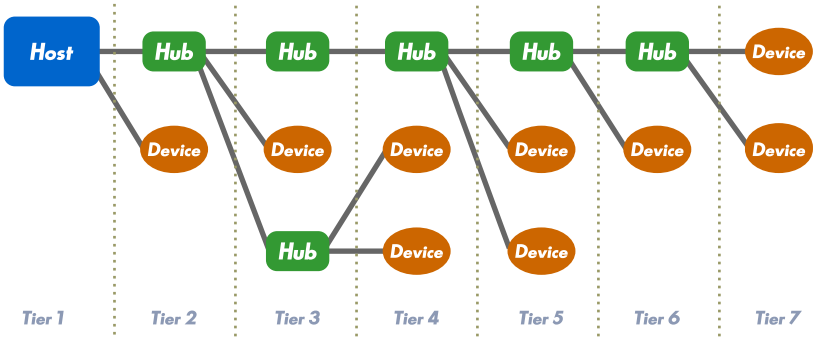

Figure 1: *Sample USB Bus Topology.*
A USB bus can only have a single USB host device. This host can support up to 127 different devices on a single port. There is an upper limit of 7 tiers of devices which means that a maximum of 5 hubs can be connected inline.

The USB bus has a tiered star topology. At the root tier is the USB host. All devices connect to this device either directly or via a hub. According to the USB spec, a USB host can only support a maximum of seven tiers between a the host and a USB device.

All USB devices are connected by a four wire USB cable. These four wires are V_{BUS}, GND and the twisted pair: D+ and D-. USB uses differential signaling on the D+ and D- to encode binary information. The idle polarity of the signal indicates whether the device is a low/full speed or high speed device.

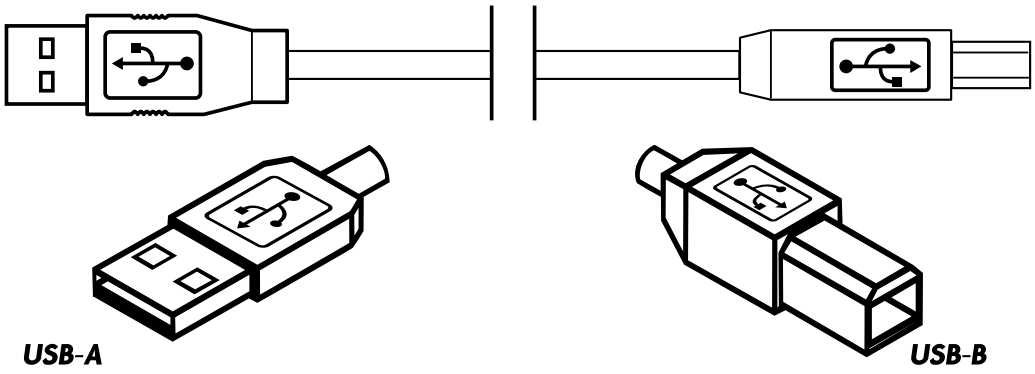

Figure 2: *USB Cable*
A USB cable has two different types of connectors: A and B. A connectors connect upstream towards the Host and B connectors connect downstream to the Devices.

USB cables have two different types of connectors: A and B. At the most basical level, A type connectors connect a device to the host or in the host direction and the B connectors

connect to downstream devices. This is part of the spec to prevent loopbacks in the USB bus. The USB spec has been expanded to include a Mini-B connector to support small USB devices and the OTG spec has introduced a mini AB connector to allow for device to device connections.

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 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.

Device Class

The USB-IF Device Working Group defines a discreet number of device classes. The idea was to simplifying 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.

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. Each endpoint is a unidirectional receiver or transmitter of data.

Endpoint 0 is a special endpoint that does not have a descriptor. All devices use this endpoint for standard control transfers to configure and setup the device.

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

Control

Used for device configuration

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 tranfer for time-sensitive data. There is no error checking 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 chunks of data. This type of transfer has 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.

Enumeration and Descriptors

When a device is plugged into a host PC, the device undergoes Enumeration. Essentially this means that the host recognizes the presence of the device and assigns it a unique 7-bit 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.

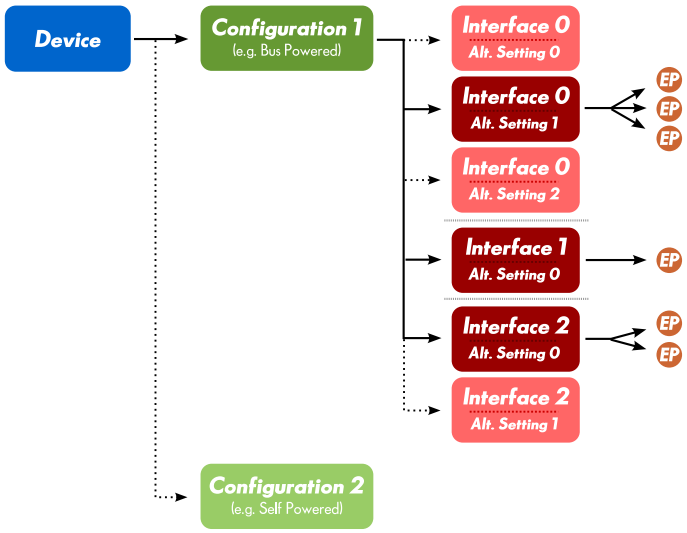

***Figure 3:** USB Descriptors*
Hierarchy of descriptors of a USB device. A device has a single Device descriptor. The Device descriptor can have multiple Configuration descriptors, but only a single one can be active at a time. The Configuration descriptor can define one or more Interface descriptors. Each of the Interface descriptors can have one or more alternate settings, but only one setting can be active at a time. The Inteface 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 heirarchy (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 endpoing 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 baove 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.

Tokens and Packets

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

All communication on the USB bus 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 will receive data to be transmitted from the addressed dev/ep.
OUT	the host will transmit data to the addressed dev/ep as receiver.
SETUP	the host will transmit control information to the device.

In the data phase, the transmitter will send one or more Data Packets. It is also possible for a device to send a NAK or STALL packet at this time indicating that it isn't able to service the IN 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.

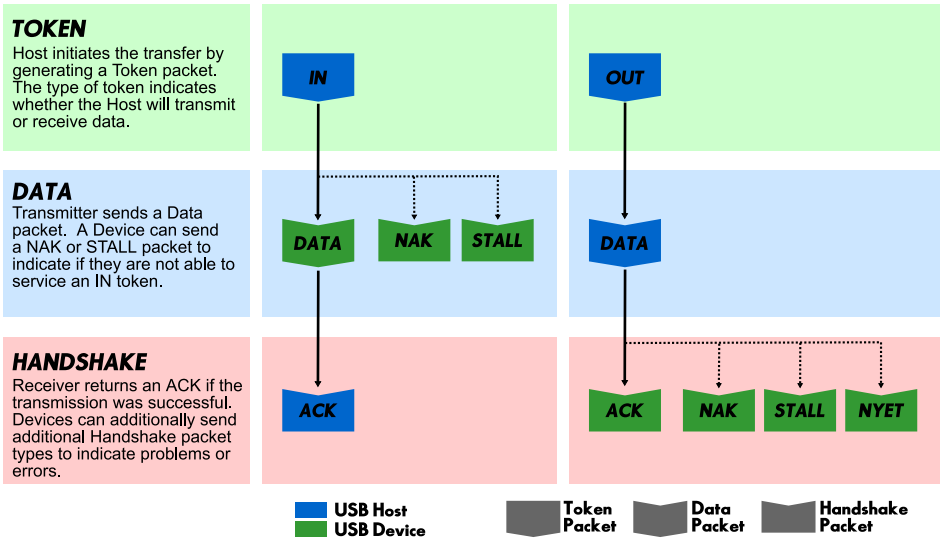

Figure 4: The Three Phases of a USB Transfer
A USB transaction has three phases

Packets

All USB packets are sent LSB first and MSB last. All packets begin with a SYNC field. It is the Start of Packet (SOP) marker and is also used to synchronize the incoming data from the transmitter with the local clock of the receiver. This SYNC field is 8 bits for full/low speed and 32 bits for high speed. The boundary between the SYNC field and the PID field are delimited by a 2-bit marker at the end of the SYNC field.

The PID field encodes the packet type. It is an 8-bit field that consists of the 4-bit packet type followed by a 4-bit one's complement of the PID packet type as a check field. If a recieved 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 1.

The format of the IN, OUT, and SETUP Token packets is shown in figure 5. The format of the SOF packet is shown in figure 6. The format of the Data packets is shown in figure 7. Lastly, the format of the Handshake packets is shown in figure 8.

Figure 5: Token Packet Format

References

- USB Implementers' Forum

Table 1: USB Packet Types

Type	PID Type	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 or the transmitter could not send data
	STALL	Endpoint halted or control pipe request is not supported
	NYET	No response yet
Special	PRE	
	ERR	
	SPLIT	
	PING	

Figure 6: Start-Of-Frame (SOF) Packet Format

Figure 7: Data Packet Format

Figure 8: Handshake Packet Format

1.2 I²C Background

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).

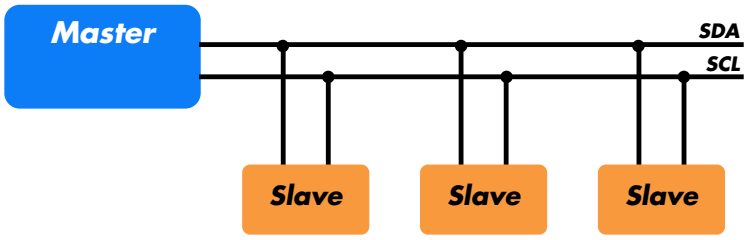

Figure 9: *Sample I²C Implementation.*
Regardless of how many slave units are attached to the I²C bus, there are only two signals connected to all of them. Consequently, there is additional overhead because an addressing mechanism is required for the master device to communicate with a specific slave 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.

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 10 shows a sample bitstream of the I²C protocol.

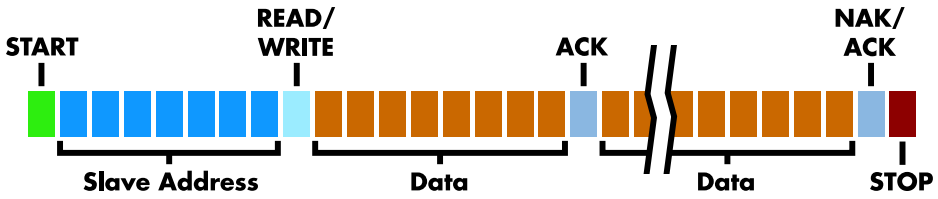

Figure 10: *I²C Protocol.*
Since there are only two wires, this protocol includes the extra overhead of the addressing and acknowledgement mechanisms.

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.

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.

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

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.

SPI Theory of Operation

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

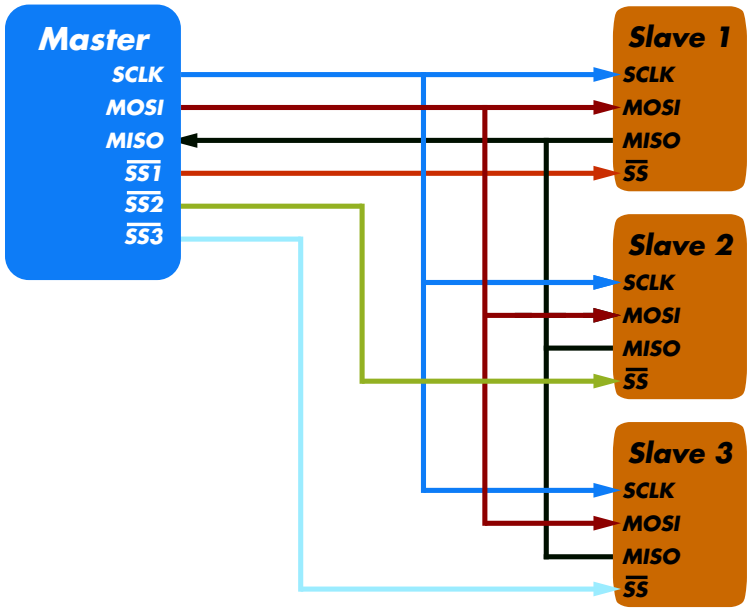

Figure 11: *Sample SPI Implementation.*
Each slave device requires a separate slave select signal (SS). This means that as devices are added, the circuit increases in complexity.

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.

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 12.

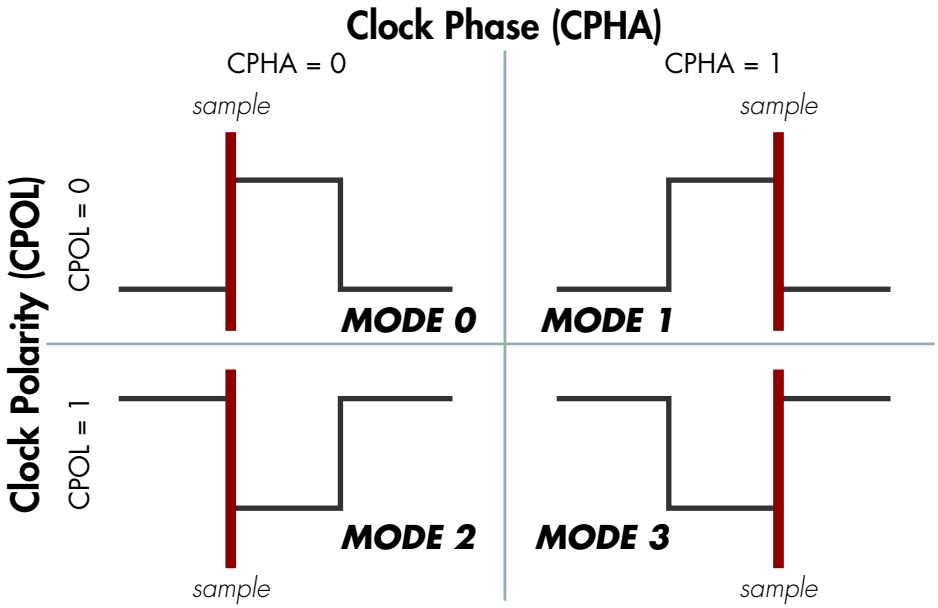

Figure 12: *SPI Modes*
The frame of the data exchange is described by two parameters, the clock polarity (CPOL) and the clock phase (CPHA). This diagram shows the four possible states for these parameters and the corresponding mode in SPI.

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.

SPI References

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

2 Hardware Specifications

2.1 Beagle USB Protocol Analyzer

Connector Specification

On one side of the Beagle USB monitor is a single USB-B receptacle. This is the **Host** side (Figure 13). This port connects to the analysis computer that is running the Beagle Data Center GUI.

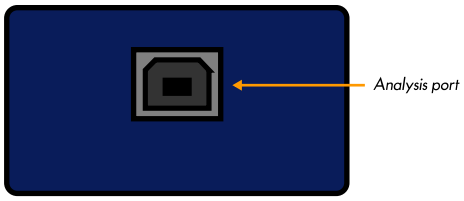

Figure 13: *Beagle USB Protocol Analyzer - Host Side*

On the opposite side is the **Target** side (Figure 14), 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.

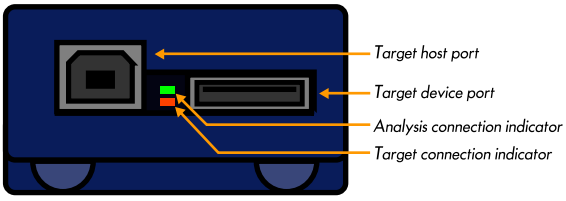

Figure 14: *Beagle USB Protocol Analyzer - Target Side*

The **Target** side acts as a USB pass-through. In order to remain within the USB 2.0 specifications, no more than 15m of USB cable should be used in total between the target host computer and the target device. The Beagle USB monitor is galvanically isolated from the USB bus to ensure the signal integrity.

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

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

Signal Specifications / Power Consumption

ESD protection

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

Speed

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

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 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.

2.2 Beagle I^2C/SPI Protocol Analyzer

Connector Specification

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

Alternatively, a split cable is available which connects to the ribbon cable and provides individual leads for each pin.

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 15), pin 1 is in the top left corner and pin 10 is in the bottom right corner.

If you flip your Beagle analyzer over (figure 16) 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.

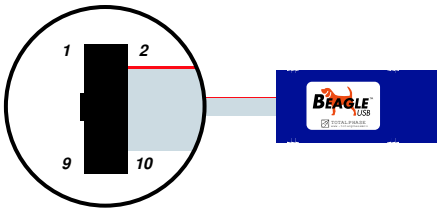

Figure 15: *The Beagle I2C/SPI Protocol Analyzer in the upright position.*
Pin 1 is located in the upper left corner of the connector and Pin 10 is located in the lower right corner of the connector.

Order of Leads

1. SCL
2. GND
3. SDA
4. NC/+5V
5. MISO
6. NC/+5V
7. SCLK
8. MOSI
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.

I^2C Pins

SCL (Pin 1):

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

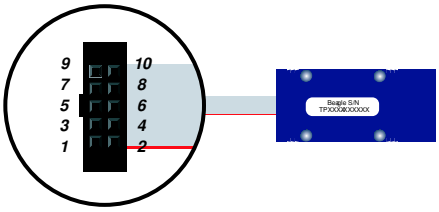

Figure 16: *The Beagle I2C/SPI Protocol Analyzer in the upside down position.*
Pin 1 is located in the lower left corner of the connector and Pin 10 is located in the upper right corner of the connector.

SDA (Pin 3):

Serial Data line – the 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.

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.

Signal Specifications / Power Consumption

Logic High Levels

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

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 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.

2.3 USB 2.0

The Beagle analyzer is a High-Speed USB 2.0 device. It requires a High-Speed USB 2.0 host controller for the analysis data connection.

2.4 Temperature Specifications

The Beagle device is 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 device outside the room temperature specification will void the hardware warranty.

3 Software

3.1 Compatibility

Linux

The Beagle software is compatible with all standard distributions of Linux with integrated USB support. Kernel 2.6 or greater is required.

Windows

The Beagle software is compatible with Windows 2000 SP4 and Windows XP SP2.

3.2 Linux USB Driver

The Beagle communications layer under Linux does not require a specific kernel driver to operate. It does however require that `/proc/bus/usb` is mounted on the system which is the case on most standard distributions.

There are two different ways to access the Beagle analyzer, through USB hotplug or by mounting the entire USB filesystem as world writable. The hotplug method is the preferred method because only the Beagle analyzer would be world writable and hence more secure.

USB Hotplug

USB hotplug requires two configuration files which are available for download from the Total Phase web site. 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).

You may now skip the following section.

World-Writable USB Filesystem

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.

3.3 Windows USB Driver

Driver Installation

On the Windows platform, the Beagle software uses a version of the libusb-win32 open source driver to access the Beagle device. For more information on this driver, please refer to the README.txt that is included with the driver. To install the appropriate USB communication driver under Windows, step through the following instructions. This is only necessary for the very first Beagle analyzer that is plugged into the PC. Subsequent plugs and unplugs should be automatically handled by the operating system.

Windows 2000:

1. When you plug in the Beagle analyzer into your PC for the first time, Windows will present the "Found New Hardware Wizard." Select "Next."
2. On the next dialog window, select "Search for a suitable driver for my device (recommended)" and click "Next."
3. On the third screen, uncheck all settings and check "Specify a location" and click "Next."
4. Click "Browse. . .", navigate to either the CD–ROM ("\usb-drivers\windows" directory), or temporary directory where the driver files have been unpacked (for downloaded updates).
5. Select "beagle.inf" and click "Open", then click "OK."
6. Click "Next" on the subsequent screen, followed by "Finish" to complete the installation. This completes the installation of the USB driver.

Windows XP:

1. When you plug in the Beagle analyzer into your PC for the first time, Windows will present the "Found New Hardware Wizard."
2. Select "Install from a list or specific location (Advanced)" and click "Next."

3. Select "Search for best driver in these locations:", uncheck "Search removable media", check "Include this location in the search."
4. Click "Browse...", expand My Computer and navigate to either the CD–ROM ("\usb-drivers\windows" directory), or temporary directory where the driver files have been unpacked (for downloaded updates).
5. Click "OK", then click "Next."
6. A dialog will inform the user that the USB driver has been installed. Click "Finish."

Both Windows 2000 and Windows XP:

7. Once the installation is complete, confirm that the installation was successful by checking that the device appears in the "Device Manager." To navigate to the "Device Manager" screen select "Control Panel | System Properties | Hardware | Device Manager."
8. The Beagle device should appear under the "LibUSB-Win32 Devices" section.

Driver Removal

Ordinarily, there is usually no harm in leaving the Beagle USB drivers installed in the operating system. However, if it is necessary that the drivers be removed, please follow the steps outlined below.

1. Plug in the Beagle device whose driver you wish to uninstall.
2. Navigate to the "Device Manager" screen by selecting "Control Panel | System Properties | Hardware | Device Manager."
3. Right click on the Beagle device which should appear under the "LibUSB-Win32 Devices" section.
4. Open the properties dialog.
5. Select the "Driver" tab and choose "Uninstall."
6. Repeat steps 1–5 for each different type (USB, I2C/SPI) of Beagle device you wish to uninstall.
7. Now use the file searching feature of Windows to search in `c:\WINNT\inf` for all files containing the text "Beagle."
8. Delete all files with the extension "`.inf`".

3.4 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 $n-1$.

Detecting Ports

To determine the ports to which the Beagle analyzers have been assigned, use the `beagle_find_devices` routine as described in following API documentation.

3.5 Beagle Dynamically Linked Library

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, or USB through a Beagle device. 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).

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 Linux system this is as follows:

1. First, search for the shared object in the application binary path. Note, that this step requires `/proc` filesystem support, which is standard in 2.4.x kernels. If the `/proc` filesystem is not present, this step is skipped.
2. Next, search in the application's 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 Windows system, this is as follows:

1. The directory from which the application binary was loaded.
2. The application's current directory.
3. 32–bit system directory. (Ex: c:\winnt\System32) [Windows NT/2000/XP only]

4. 16–bit system directory. (Ex: c:\winnt\System or c:\windows\system)
5. The windows directory. (Ex: c:\winnt or c:\windows)
6. The directories listed in the PATH environment variable.

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

DLL Versioning

The Beagle DLL checks to ensure that the firmware of a given Beagle device 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.

3.6 Rosetta Language Bindings: API Integration into Custom Applications

Overview

The Beagle Rosetta language bindings make integration of the Beagle API into custom applications simple. Accessing Beagle 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).

The 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 C and C++ are supported for the Beagle. The integration for the C language bindings is described below.

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.

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, `BEAGLE_INCOMPATIBLE_LIBRARY`.

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.

3.7 Application Notes

Receive Saturation

Once enabled, the Beagle adapter 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 PC host, 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 device 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.

Threading

The Cheetah 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 Cheetah functionality, each Cheetah 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 Cheetah open and close operations are thread-safe and cannot happen concurrently with any other Cheetah operations. However, once a Cheetah device is opened, all operations to that device can be dispatched to a separate thread as long as no other threads access that same Cheetah device.

4 Firmware

4.1 Philosophy

The firmware included with the Beagle analyzer provides for the analysis of the supported protocols. It is installed at the factory during manufacturing. Updates to this firmware are provided through a device upgrade utility. The Beagle software automatically detects firmware compatibility and will inform the user if an upgrade is required.

4.2 Procedure

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

5 API Documentation

5.1 Introduction

The API documentation describes the Beagle Rosetta C bindings.

5.2 General Data Types

The following definitions are provided for convenience. The Beagle API provides both signed and unsigned data types.

```
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;
```

5.3 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, `BEAGLE_UNABLE_TO_LOAD_LIBRARY` and `BEAGLE_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 `BEAGLE_UNABLE_TO_LOAD_DRIVER` or `BEAGLE_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 (`BEAGLE_HEADER_VERSION` found in `beagle.h` and `BEAGLE_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.

Any API function that accepts a Beagle handle can potentially return the error code `BEAGLE_INVALID_HANDLE` if the handle does not correspond to a valid Beagle device that has already been opened. If this error is received, check the application code to ensure that the `beagle_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 device can return the error `BEAGLE_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, or USB subsystems have been disabled by beagle_disable, all other API functions that interact with I^2C , SPI, and USB will return `BEAGLE_I2C_NOT_ENABLED`, `BEAGLE_SPI_NOT_ENABLED`, or `BEAGLE_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.

5.4 General

Interface

Find Devices (beagle_find_devices)

```
int beagle_find_devices (int      nelem,
                         u16 * devices);
```

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

Arguments

`nelem`: Maximum size of the array

`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 OR'ed with `BEAGLE_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 (beagle_find_devices_ext)

```
int beagle_find_devices_ext (int      nelem,
                             u16 * devices,
                             u32 * unique_ids);
```

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

Arguments

`nelem`: Maximum size of the array

`devices`: array into which the port numbers are returned

`unique_ids`: array into which the unique IDs are returned

Return Value

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

Specific Error Codes

None.

Details

This function is the same as `beagle_find_devices()` except that is 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.

Open a Beagle device (beagle_open)

```
Beagle beagle_open (int port_number);
```

Open the Beagle port.

Arguments

`port_number`: The Beagle device port number. This port number is the the same as the one obtained from the `beagle_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

`BEAGLE_UNABLE_TO_OPEN`: The specified port is not connected to a Beagle device or the port is already in use.

`BEAGLE_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 beagle_open_ext() 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 `beagle_open_ext()` is recommended.

Open a Beagle device (beagle_open_ext)

```
Beagle beagle_open_ext (int port_number, BeagleExt *beagle_ext);
```

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

Arguments

`port_number`: same as `beagle_open`

`beagle_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

`BEAGLE_UNABLE_TO_OPEN`: The specified port is not connected to a Beagle device or the port is already in use.

BEAGLE_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 beagle_ext.

Details

If 0 is passed as the pointer to the structure, this function will behave exactly like beagle_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 device. See the API function beagle_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 and hardware versions. */
    u16 software;
    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;

   /*
    * 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 `beagle_open()` is recommended.

This open function also terminates all slave functionality as described for the `beagle_open()` call.

Close a Beagle connection (beagle_close)

```
int beagle_close (Beagle beagle);
```

Close the Beagle port.

Arguments

`beagle`: handle of a Beagle analyzer to be closed

Return Value

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

Specific Error Codes

None.

Details

None.

Get Features (beagle_features)

```
int beagle_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 device are returned. These are a bit–mask of the following values.

```
#define BEAGLE_FEATURE_NONE         (0)
#define BEAGLE_FEATURE_I2C          (1<<0)
#define BEAGLE_FEATURE_SPI          (1<<1)
#define BEAGLE_FEATURE_USB          (1<<2)
```

Specific Error Codes

None.

Details

None.

Get Features by Unique ID (beagle_unique_id_to_features)

```
int beagle_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 device are returned. See beagle_features for details on the bit map.

Specific Error Codes

None.

Details

None.

Get Unique ID (beagle_unique_id)

```
u32 beagle_unique_id (Beagle beagle);
```

Return the unique ID of the given Beagle device.

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 (beagle_status_string)

```
const char *beagle_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 (beagle_version)

```
int beagle_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 `BEAGLE_OK` on success.

Specific Error Codes

None.

Details

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

See the details of `beagle_open_ext` for the definition of `BeagleVersion`.

Capture Latency (beagle_latency)

```
int beagle_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 `BEAGLE_OK` on success.

Specific Error Codes

None.

Details

Set the capture latency to the specified number of milliseconds. The capture latency is the minimum amount of time a read call will block until it can process data from the Beagle analyzer.

Setting this parameter high will increase the size of the buffers used to capture data from the Beagle analyzer. Larger buffers incur larger latencies. Setting this parameter to a smaller value will decrease the buffer size but may result in lost data.

Timeout Value (beagle_timeout)

```
int beagle_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 `BEAGLE_OK` on success.

Specific Error Codes

None.

Details

Set the idle timeout to the specified number of milliseconds.

If a read call is made and there has not been any new data for the last specified timeout interval, that call will return with a status of BEAGLE_TIMEOUT and the number of bytes returned will be fewer than requested.

This setting is distinctly different than the latency setting.

If the timeout is set to 0, there is no timeout and the read call will block until a full packet is received.

Sleep (beagle_sleep_ms)

```
u32 beagle_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 (beagle_target_power)

```
int beagle_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 2.

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.

Table 2: `power_flag` *enumerated types*

`BEAGLE_TARGET_POWER_OFF`	Disable target power pin
`BEAGLE_TARGET_POWER_ON`	Enable target power pin
`BEAGLE_TARGET_POWER_QUERY`	Queries the target power pin state

Specific Error Codes

`BEAGLE_FUNCTION_NOT_AVAILABLE`: The hardware version is not compatible with this feature. Only the Beagle I^2C /SPI monitor supports switchable target power pins.

Details

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 is normally passively monitoring the bus.

Host Interface Speed (beagle_host_ifce_speed)

```
int beagle_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 host computer is communicating with the given Beagle device. See Table 3.

Table 3: `interface speed` *enumerated types*

`BEAGLE_HOST_IFCE_FULL_SPEED`	Full speed (12Mbps) interface
`BEAGLE_HOST_IFCE_HIGH_SPEED`	High Speed (480Mbps) interface

Specific Error Codes

None.

Details

Used to determine the USB communication rate between the Beagle device and the host. The Beagle analyzers require a High Speed USB connection with the host. Capturing from a Beagle device that is connected at USB Full Speed or Low Speed can cause data to be lost and corruption of capture data.

Benchmarking

Available Read Buffers (beagle_buffers_avail)

```
int beagle_buffers_avail (Beagle beagle);
```

Query the number of read buffers available.

Arguments

`beagle`: handle of a Beagle analyzer

Return Value

The number of available USB read buffers.

Specific Error Codes

None.

Details

USB read buffers are used by the analysis computer to receive the incoming data from the Beagle device. Calling this function will return the number of buffers currently available to receive data. If the number of available USB read buffers drops to zero, capture data from the device may be lost.

Buffer Information (beagle_buffers_info)

```
int beagle_buffers_info (Beagle beagle, int *max_buffers, int *buffer_size);
```

Query the buffer capacity of the driver.

Arguments

`beagle`: handle of a Beagle analyzer.

`max_buffers`: maximum number of buffers as defined by the driver.

`buffer_size`: the buffer size calculated by the driver to fulfill the latency requirements.

Return Value

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

Specific Error Codes

None.

Details

Used for checking the current performance tuning the Beagle interface.

Communication Speed Benchmark (beagle_commtest)

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

Test the Beagle 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 computer's ability to process data received from the Beagle analyzer. The function commands the given Beagle device to send test packets at the given frequency (see `beagle_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.

Monitoring API

Enable Monitoring (beagle_enable)

```
int beagle_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 4)

Table 4: `protocol` *definitions*

`BEAGLE_PROTOCOL_NONE`	No Protocol
`BEAGLE_PROTOCOL_COMMTEST`	Comm Tester
`BEAGLE_PROTOCOL_USB`	USB Protocol
`BEAGLE_PROTOCOL_I2C`	I^2C Protocol
`BEAGLE_PROTOCOL_SPI`	SPI Protocol

Return Value

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

Specific Error Codes

`BEAGLE_FUNCTION_NOT_AVAILABLE`: The connected Beagle device does not support capturing for the requested protocol.

`BEAGLE_UNKNOWN_PROTOCOL`: A protocol was requested that does not appear in the enumeration detailed in Table 4.

Details

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

Stop Monitoring (beagle_disable)

```
int beagle_disable (Beagle beagle);
```

Stop monitoring of packets.

Arguments

`beagle`: handle of a Beagle analyzer

Return Value

A Beagle status code of BEAGLE_OK is returned on success.

Specific Error Codes

None.

Details

Stops monitoring on the given Beagle device.

Sample Rate (beagle_samplerate)

```
int beagle_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

BEAGLE_FUNCTION_NOT_AVAILABLE: The Beagle device does not support changing the sample rate.

BEAGLE_STILL_ACTIVE: Monitoring is active and the sample rate cannot be changed.

Details

Changes the sample rate for a Beagle device. 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 analyzer does not support changing the sample rate, so it will always return the current sample rate.

5.5 Port-specific APIs

Introduction

All read functions return a status value through the status parameter. Table 5 provides a listing of all the possible values that may be returned.

Table 5: `read status` *definitions*

BEAGLE_READ_OK	Read successful.
BEAGLE_READ_TIMEOUT	No data was seen before the timeout interval occurred.
BEAGLE_READ_ERR_MIDDLE_OF_PACKET	Data collection was started in the middle of a packet.
BEAGLE_READ_ERR_SHORT_BUFFER	The packet was longer than the buffer size.
BEAGLE_READ_ERR_PARTIAL_LAST_BYTE	The last byte in the buffer is incomplete.

I²C API

I²C Pullups (beagle_i2c_pullup)

```
int beagle_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. See 6.

Table 6: `protocol` *definitions*

BEAGLE_I2C_PULLUP_OFF	Disable the pullup resistors.
BEAGLE_I2C_PULLUP_ON	Enable the pullup resistors.
BEAGLE_I2C_PULLUP_QUERY	Query the status of the pullup resistors.

Return Value

A Beagle status code of BEAGLE_OK is returned on success. If the function is BEAGLE_I2C_PULLUP_QUERY, the state of the pullups is returned.

Specific Error Codes

BEAGLE_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 (beagle_i2c_read)

```
int beagle_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 table 5

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

Return Value

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

Specific Error Codes

None.

Details

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 rate clock.

Ordinarily the number of bytes read will equal the requested number of bytes.

Read I²C with byte-level timing (beagle_i2c_read_data_timing)

```
int beagle_i2c_read_data_timing (Beagle  beagle,
                                 u32 *   status,
                                 u64 *   time_sop,
                                 u64 *   time_duration,
                                 u32 *   time_dataoffset,
                                 int     max_bytes,
                                 u16 *   data_in,
                                 u32 *   data_timing);
```

Read data from the I²C port.

Arguments

beagle: handle of a Beagle analyzer

status: filled with the status bitmask as detailed in table 5

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

`max_bytes`: maximum number of bytes to read

`data_in`: an allocated array of `u16` which is filled with the received data

`data_timing`: an allocated array of `u32` which is filled with timing data for each byte read

Return Value

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

Specific Error Codes

None.

Details

The data_in and data_timing arrays should be allocated to max_bytes.

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

Ordinarily the number of bytes read will equal the requested number of bytes.

Read I²C with bit-level timing (beagle_i2c_read_bit_timing)

```
int beagle_i2c_read_bit_timing (Beagle  beagle,
                                u32 *   status,
                                u64 *   time_sop,
                                u64 *   time_duration,
                                u32 *   time_dataoffset,
                                int     max_bytes,
                                u16 *   data_in,
                                u32 *   bit_timing);
```

Read data from the I²C port.

Arguments

`beagle`: handle of a Beagle analyzer

`status`: filled with the status bitmask as detailed in table 5

`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

`max_bytes`: maximum number of bytes to read

`data_in`: an allocated array of `u16` which is filled with the received data

`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

The data_in array should be allocated to max_bytes. The bit_timing array should be allocated to max_bytes * 10, to allow for start and stop bits.

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

Ordinarily the number of bytes read will equal the requested number of bytes.

SPI API

SPI Configuration (beagle_spi_configure)

```
int beagle_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 7

sck_sampling_edge: sets data sampling on the leading or trailing edge of the clock signal, see table 8

bitorder: sets big-endian or little-endian bit order, see table 9

Table 7: SPI Polarity *definitions*

BEAGLE_SS_ACTIVE_LOW	Set active low polarity
BEAGLE_SS_ACTIVE_HIGH	Set active high polarity

Table 8: SPI Sampling Edge *definitions*

BEAGLE_SPI_SAMPLING_EDGE_RISING	Sample on the leading edge
BEAGLE_SPI_SAMPLING_EDGE_FALLING	Sample on the trailing edge

Table 9: SPI Bit Order *definitions*

BEAGLE_SPI_BITORDER_MSB	Big-endian bit ordering
BEAGLE_SPI_BITORDER_LSB	Little-endian bit ordering

Return Value

A Beagle status code of BEAGLE_OK is returned on success.

Specific Error Codes

BEAGLE_STILL_ACTIVE: The receiver must be disabled to change the configuration.

BEAGLE_FUNCTION_NOT_AVAILABLE: The hardware version is not compatible with this feature. Only the I^2C /SPI device supports SPI configuration.

Details

The SPI standard is much more loosely defined than I^2C 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 (beagle_spi_read)

```
int beagle_spi_read (Beagle  beagle,
                     u32 *   status,
                     u64 *   time_sop,
                     u64 *   time_duration,
                     u32 *   time_dataoffset,
                     int     max_bytes,
                     u08 *   data_mosi,
                     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 5

`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

`max_bytes`: maximum number of bytes to read

`data_mosi`: an allocated array of `u08` which is filled with the data sent from the master to the slave

`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 `data_mosi` and `data_miso` pointers should be allocated at least as large as `max_bytes`.

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

Ordinarily the number of bytes read will equal the requested number of bytes.

Read SPI with byte-level timing (beagle_spi_read_data_timing)

```
int beagle_spi_read_data_timing (Beagle  beagle,
                                 u32 *   status,
                                 u64 *   time_sop,
                                 u64 *   time_duration,
                                 u32 *   time_dataoffset,
```

```
                                    int     max_bytes,
                                    u08 *   data_mosi,
                                    u08 *   data_miso,
                                    u32 *   data_timing);
```

Read data from the SPI port.

Arguments

`beagle`: handle of a Beagle analyzer

`status`: filled with the status bitmask as detailed in table 5

`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

`max_bytes`: maximum number of bytes to read

`data_mosi`: an allocated array of `u08` which is filled with the data sent from the master to the slave

`data_miso`: an allocated array of `u08` which is filled with the data sent from the slave to the master

`data_timing`: an allocated array of `u32` which is filled with timing data for each byte read

Return Value

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

Specific Error Codes

None.

Details

The `data_mosi`, `data_miso` and `data_timing` pointers should be allocated at least as large as `max_bytes`.

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

Ordinarily the number of bytes read will equal the requested number of bytes.

Read SPI with bit-level timing (beagle_spi_read_bit_timing)

```
int beagle_spi_read_bit_timing (Beagle  beagle,
                                u32 *   status,
                                u64 *   time_sop,
                                u64 *   time_duration,
                                u32 *   time_dataoffset,
                                int     max_bytes,
                                u08 *   data_mosi,
                                u08 *   data_miso,
                                u32 *   bit_timing);
```

Read data from the SPI port.

Arguments

`beagle`: handle of a Beagle analyzer

status: filled with the status bitmask as detailed in table 5

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

max_bytes: maximum number of bytes to read

data_mosi: an allocated array of u08 which is filled with the data sent from the master to the slave

data_miso: an allocated array of u08 which is filled with the data sent from the slave to the master

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

The data_mosi, data_miso pointers should be allocated at least as large as max_bytes.

The bit_timing pointer should be allocated to max_bytes * 8.

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

Ordinarily the number of bytes read will equal the requested number of bytes.

USB API

Read USB (beagle_usb_read)

```
int beagle_usb_read (Beagle  beagle,
                     u32 *   status,
                     u64 *   time_sop,
                     u64 *   time_duration,
                     u32 *   time_dataoffset,
                     int     max_bytes,
                     u08 *   packet);
```

Read data from the USB port.

Arguments

beagle: handle of a Beagle analyzer

status: filled with the status bitmask as detailed in table 5

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

max_bytes: maximum number of bytes to read

packet: an allocated array of u08 which is filled with the received data

Return Value

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

Specific Error Codes

None.

Details

The packet pointer should be allocated at least as large as num_bytes.

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

Ordinarily the number of bytes read will equal the requested number of bytes.

The first byte of the USB packet is the packet ID. An enumeration is provided that defines all the possible packet IDs in table 10.

In addition to the general read status values in table 5, there are some USB specific status values enumerated. See table 11.

Table 10: USB Packet ID *definitions*

BEAGLE_USB_PID_OUT	0xe1
BEAGLE_USB_PID_IN	0x69
BEAGLE_USB_PID_SOF	0xa5
BEAGLE_USB_PID_SETUP	0x2d
BEAGLE_USB_PID_DATA0	0xc3
BEAGLE_USB_PID_DATA1	0x4b
BEAGLE_USB_PID_DATA2	0x87
BEAGLE_USB_PID_MDATA	0x0f
BEAGLE_USB_PID_ACK	0xd2
BEAGLE_USB_PID_NAK	0x5a
BEAGLE_USB_PID_STALL	0x1e
BEAGLE_USB_PID_NYET	0x96
BEAGLE_USB_PID_PRE	0x3c
BEAGLE_USB_PID_ERR	0x3c
BEAGLE_USB_PID_SPLIT	0x78
BEAGLE_USB_PID_PING	0xb4
BEAGLE_USB_PID_RESERVE	0xf0

Table 11: USB Read Status *definitions*

BEAGLE_READ_USB_ERR_BAD_SIGNALS	0x10000
BEAGLE_READ_USB_ERR_BAD_SYNC	0x20000
BEAGLE_READ_USB_ERR_BIT_STUFF	0x40000
BEAGLE_READ_USB_ERR_FALSE_EOP	0x80000
BEAGLE_READ_USB_ERR_LONG_EOP	0x100000
BEAGLE_READ_USB_ERR_BAD_PID	0x200000
BEAGLE_READ_USB_ERR_BAD_CRC	0x400000
BEAGLE_READ_USB_HOST_DISCONNECT	0x800000
BEAGLE_READ_USB_TARGET_DISCONNECT	0x1000000
BEAGLE_READ_USB_HOST_CONNECT	0x2000000
BEAGLE_READ_USB_TARGET_CONNECT	0x4000000

Read USB with byte-level timing (beagle_usb_read_data_timing)

```
int beagle_usb_read_data_timing (Beagle  beagle,
                                 u32 *   status,
                                 u64 *   time_sop,
                                 u64 *   time_duration,
                                 u32 *   time_dataoffset,
                                 int     max_bytes,
                                 u08 *   packet,
                                 u32 *   data_timing);
```

Read data from the USB port.

Arguments

beagle: handle of a Beagle analyzer

status: filled with the status bitmask as detailed in table 5

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

max_bytes: maximum number of bytes to read

packet: an allocated array of u08 which is filled with the received data

data_timing: an allocated array of u32 which is filled with timing data for each byte read

Return Value

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

Specific Error Codes

None.

Details

The data_in and data_timing arrays should be allocated to max_bytes.

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

Read USB with bit-level timing (beagle_usb_read_bit_timing)

```
int beagle_usb_read_bit_timing (Beagle  beagle,
                                u32 *   status,
                                u64 *   time_sop,
                                u64 *   time_duration,
                                u32 *   time_dataoffset,
                                int     max_bytes,
                                u08 *   packet,
                                u32 *   bit_timing);
```

Read data from the USB port.

Arguments

beagle: handle of a Beagle analyzer

status: filled with the status bitmask as detailed in table 5

`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

`max_bytes`: maximum number of bytes to read

`packet`: an allocated array of `u08` which is filled with the received data

`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

The packet array should be allocated to max_bytes. The bit_timing array should be allocated to max_bytes * 8.

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

5.6 Error Codes

Table 12: Beagle API Error Codes

Literal Name	Value	beagle_status_string() return value
BEAGLE_OK	0	ok
BEAGLE_UNABLE_TO_LOAD_LIBRARY	-1	unable to load library
BEAGLE_UNABLE_TO_LOAD_DRIVER	-2	unable to load usb driver
BEAGLE_UNABLE_TO_LOAD_FUNCTION	-3	unable to load function
BEAGLE_INCOMPATIBLE_LIBRARY	-4	incompatible library version
BEAGLE_INCOMPATIBLE_DEVICE	-5	incompatible device version
BEAGLE_INCOMPATIBLE_DRIVER	-6	incompatible driver version
BEAGLE_COMMUNICATION_ERROR	-7	communication error
BEAGLE_UNABLE_TO_OPEN	-8	unable to open device
BEAGLE_UNABLE_TO_CLOSE	-9	unable to close device
BEAGLE_INVALID_HANDLE	-10	invalid device handle
BEAGLE_CONFIG_ERROR	-11	configuration error
BEAGLE_UNKNOWN_PROTOCOL	-12	unknown beagle protocol
BEAGLE_STILL_ACTIVE	-13	beagle still active
BEAGLE_FUNCTION_NOT_AVAILABLE	-14	beagle function not available
BEAGLE_COMMTEST_NOT_AVAILABLE	-100	comm test feature not available
BEAGLE_COMMTEST_NOT_ENABLED	-101	comm test not enabled
BEAGLE_I2C_NOT_AVAILABLE	-200	i2c feature not available
BEAGLE_I2C_NOT_ENABLED	-201	i2c not enabled
BEAGLE_SPI_NOT_AVAILABLE	-300	spi feature not available
BEAGLE_SPI_NOT_ENABLED	-301	spi not enabled
BEAGLE_USB_NOT_AVAILABLE	-400	usb feature not available
BEAGLE_USB_NOT_ENABLED	-401	usb not enabled

6 Legal / Contact

6.1 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.

6.2 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.

6.3 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.

List of Figures

List of Tables

Contents