**Cheetah SPI Host Adapter User Manual**

1 General Overview
1.1 SPI Background
2 Hardware Specifications
2.1 Connector Specification
2.2 Signal Specifications / Power Consumption
2.3 USB 2.0
2.4 Temperature Specifications
2.5 SPI Signaling Characteristics
3 Software
3.1 Compatibility
3.2 Windows USB Driver
3.3 Linux USB Driver
3.4 Mac OS X USB Driver
3.5 USB Port Assignment
3.6 Cheetah Dynamically Linked Library
3.7 Rosetta Language Bindings: API Integration into Custom Applications
3.8 Application Notes
4 Firmware
4.1 Philosophy
4.2 Procedure
5 API Documentation
5.1 Introduction
5.2 General Data Types
5.3 Notes on Status Codes
5.4 General
5.5 SPI Interface
5.6 Error Codes
6 Legal / Contact
6.1 Disclaimer
6.2 Life Support Equipment Policy
6.3 Contact Information

# 1 General Overview

Top

## 1.1 SPI Background

### 1.1.1 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.

### 1.1.2 SPI Theory of Operation

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

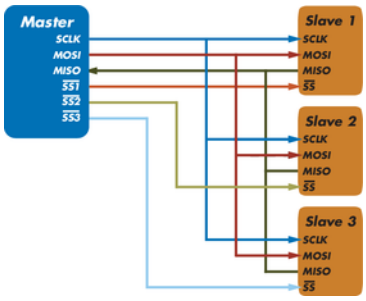



***Figure 1** : Sample SPI Implementation.*
*Each slave device requires a separate slave select signal (SS). This means that as devices are added, the circuit increases in complexity.*

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 slaves 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 slaves presence.

### 1.1.3 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 2.

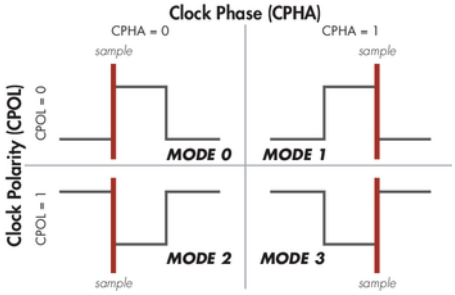



***Figure 2** : SPI Modes*
*The frame of the data exchange is described by two parameters, the clock polarity (CPOL) and the clock phase (CPHA). This diagram shows the four possible states for these parameters and the corresponding mode in SPI.*

### 1.1.4 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.

### 1.1.5 SPI References

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

## 2 Hardware Specifications

Top

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

### 2.1.1 Orientation

The ribbon cable pin order follows the standard convention. The red line indicates the first position. When looking at your Cheetah adapter in the upright position (Figure 3 ), pin 1 is in the top left corner and pin 10 is in the bottom right corner.

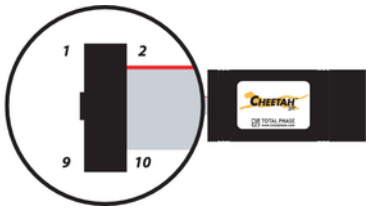



***Figure 3*** *: The Cheetah SPI Host Adapter in the upright position.*
*Pin 1 is located in the upper left corner of the connector and Pin 10 is located in the lower right corner of the connector.*

If you flip your Cheetah adapter over (Figure 4) 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.

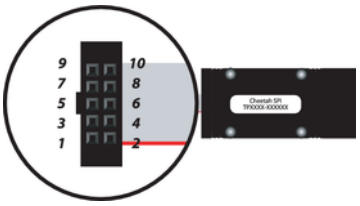



***Figure 4*** *: The Cheetah SPI Host Adapter in the upside down position.*
*Pin 1 is located in the lower left corner of the connector and Pin 10 is located in the upper right corner of the connector.*

### 2.1.2 Order of Leads

1. SS2
2. GND
3. SS3
4. NC/+5V
5. MISO
6. NC/+5V
7. SCLK
8. MOSI
9. SS1
10. GND

### 2.1.3 Ground

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

It is imperative that the Cheetah adapter'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.

### 2.1.4 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.

**SS1 (Pin 9):**

Primary Slave Select – the primary control line that allows slaves to be turned on and off via hardware control. (This SS is in the same location as the SS line of the Aardvark and Beagle products.)

**SS2 (Pin 1):**

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

**SS3 (Pin 3):**

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

#### 2.1.5 Powering Downstream Devices

It is possible to power a downstream target, such as an SPI EEPROM with the Cheetah adapter's power (which is provided by the USB bus). It is ideal if the downstream device does not consume more than 20-30 mA. The Cheetah adapter 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 Cheetah adapter 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 Cheetah adapter always reports itself to the host as a low-power device. Therefore, drawing large amounts of current from the host is not advisable.

Top

## 2.2 Signal Specifications / Power Consumption

#### 2.2.1 Logic High Levels

All signal levels should be nominally 3.3 volts (+/- 10%) logic high. This allows the Cheetah adapter 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.

#### 2.2.2 ESD protection

The Cheetah adapter 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 SPI bus.

#### 2.2.3 Power Consumption

The Cheetah adapter consumes approximately 125 mA of power from the host PC. However, it reports itself to the host PC as a low-power device. This reporting allows the Cheetah adapter to be used when its host 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 own 100 mA budget. If there are any concerns regarding the total amount of available current supply, it is advisable to plug the Cheetah adapter directly into the host PC's USB host port or to use a self-powered hub.

Top

## 2.3 USB 2.0

The Cheetah adapter is a High-Speed USB 2.0 device. It can be plugged into either a high-speed or full-speed port. However, a high-speed port must be used to achieve full throughput at high data rates (SPI clock rates >3 Mbps).

Top

## 2.4 Temperature Specifications

The Cheetah adapter 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 Cheetah device outside the room temperature specification will void the hardware warranty.

Top

## 2.5 SPI Signaling Characteristics

### 2.5.1 SPI Waveforms

The SPI signaling is characterized by the waveforms in Figures 5 and 6.

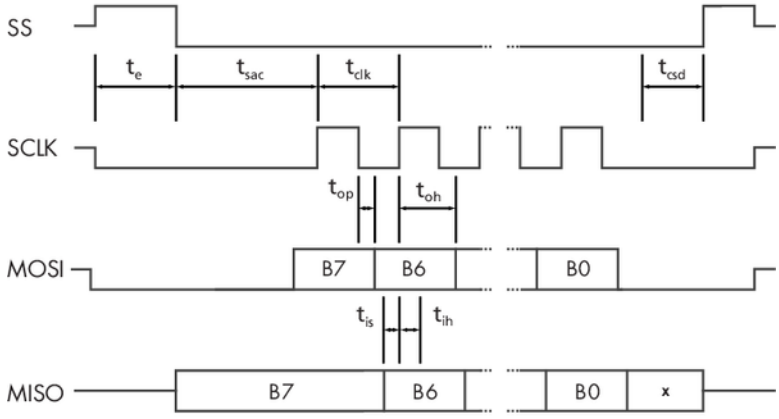



***Figure 5*** *(above): SPI Timing Requirements*

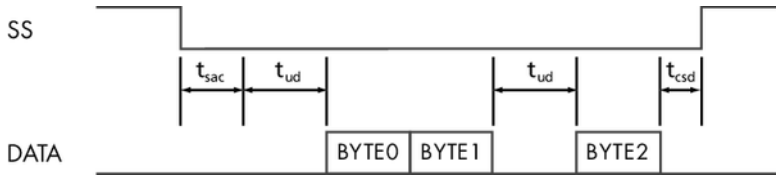



***Figure 6*** *(above): SPI Byte-Level Timing*

**Table 1** : SPI Timing Parameters

| Symbol | Parameter | Min | Max | Units |
|---|---|---|---|---|
| $t_{clk}$ | Clock period | 25 | 10000 | ns |
| $t_e$ | Output Enable to SS control delay | 8 | 8 | $t_{clk}$ |
| $t_{sac}$ | SS# assertion to first clock edge | 6.5 | 7 | $t_{clk}$ |
| $t_{csd}$ | SS# deassertion from last clock edge | 1.5 | 2 | $t_{clk}$ |
| $t_{op}$ | MOSI propagation time | 0 | 5.8 | ns |
| $t_{oh}$ | MOSI hold time | 0.5 | 0.5 | $t_{clk}$ |
| $t_{is}$ | MISO setup time | 4.8 | N/A | ns |
| $t_{ih}$ | MISO hold time | 0 | N/A | ns |
| $t_{ud}$ | User Insertable Delay | 8 | $2^{32}$ | $t_{clk}$ |

Notes:

1. The above timings only correspond to actions performed within a given SPI transaction. Actions that span transactions will be subject to inter-transaction USB delays. However, many SPI packets (delineated by SS assert/deassert) can be sent within a single transaction.
2. User insertable delays are quantized in blocks of 8 clock cycles.
3. A user delay can be inserted to stretch $t_e$, $t_{sac}$ or, $t_{csd}$ as needed.
4. There is normally no gap between data bytes, although a user delay can be inserted between bytes if the target SPI slave device needs time to process each received byte.
5. The MOSI hold time ($t_{oh}$ ) can be longer than half of one clock period, depending on the exact MOSI propagation time. However, the maximum guaranteed hold time is 0.5*$t_{clk}$.

6. The parameters $t_{sac}$ and $t_{csd}$ differ based on mode and polarity. For example, $t_{sac}$ = 6.5 and $t_{csd}$ = 2.0 clock periods for modes 0 and 2. Likewise, $t_{sac}$ = 7.0 and $t_{csd}$ = 1.5 clock periods for modes 1 and 3.

#### 2.5.2 Speeds

The Cheetah device has a flexible clock generator that can produce SPI clock rates at a very fine granularity. The minimum bit rate is 100 KHz and the maximum bit rate is 40 MHz. It can be overclocked to 50 MHz but this is not officially supported. Many intermediate bit rates are available, often with a 1-2 kHz precision.

The Cheetah software and hardware have been meticulously designed to ensure maximal average throughput over the USB bus. In other words, if the SPI clock rate is set to 30 MHz, the average data rate across an entire transaction will be nearly 30 Mbps, end-to-end from host PC to SPI target device. This property holds even for very large transactions ranging from hundreds of kilobytes to many megabytes long. This high throughput feature is only possible within a single transaction. Multiple transactions will suffer unavoidable USB bus latencies. Hence, the best throughput can be achieved for single transactions that transfer a large number of bytes at a time.

Rarely, there can be delays across the USB bus even within a transaction. While there will be a delay in the outgoing SCLK while the Cheetah adapter is paused, waiting for more data to shift out, the average throughput will not be diminished appreciably since such events happen so infrequently.

#### 2.5.3 Pin Driving

The Cheetah adapter can be connected or disconnected from the bus through software control. Namely, the device can either drive its outputs or place them in high impedance mode. When in high impedance mode the Cheetah device will hold the last value of the output lines with very weak pull-up or pull-down resistances. When connected to the bus, the Cheetah adapter persistently holds the state of the SS lines across different shift transactions. Hence, if the SS line is left asserted after one transaction, it will stay asserted until modified in a subsequent transaction.

# 3 Software

Top

## 3.1 Compatibility

Software compatibility information for the Cheetah SPI Host Adapter, including supported versions of **Windows, Linux, and macOS**, is provided on the respective software product pages under System Requirements. Be sure the device driver has been installed before plugging in the Cheetah adapter.

- Control Center Serial Software
- Flash Center Software
- Cheetah Software API

Top

## 3.2 Windows USB Driver

#### 3.2.1 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 Cheetah adapter product page on the Total Phase website.

After the driver has been installed, plugging in a Cheetah adapter for the first time will cause the adapter to be installed and associated with the correct driver. The following steps describe the feedback the user should receive from Windows after a Cheetah adapter 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 Cheetah Host Adapter" 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 Cheetah adapter should appear under the Universal Serial Bus Controllers section for Windows XP/Vista/7.

#### 3.2.2 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

Top

## 3.3 Linux USB Driver

As of version 3.08, the Cheetah communications layer under Linux no longer requires a specific kernel mode or user mode driver to operate. This differs from previous versions that required the user to ensure independently that the libusb library was installed on the system. See the README.txt in the API package for more details.

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 Cheetah adapter 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 Cheetah adapter, 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.

#### 3.3.1 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 Cheetah adapter.

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 Cheetah adapter(s)

#### 3.3.2 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: `cheetah` and `cheetah.usermap`. Please follow the following steps to enable hotplugging.

1. As superuser, unpack `cheetah` and `cheetah.usermap` to `/etc/hotplug/usb`
2. `chmod 755 /etc/hotplug/usb/cheetah`
3. `chmod 644 /etc/hotplug/usb/cheetah.usermap`
4. Unplug and replug your Cheetah adapter(s)
5. Set the environment variable `USB_DEVFS_PATH` to `/proc/bus/usb`

### 3.3.3 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 Cheetah adapter 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`

Top

## 3.4 Mac OS X USB Driver

The Cheetah communications layer under Mac OS X does not require a specific kernel driver to operate. Both Mac OS X 10.4 Tiger and 10.5 Leopard 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.

Top

## 3.5 USB Port Assignment

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

### 3.5.1 Detecting Ports

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

Top

## 3.6 Cheetah Dynamically Linked Library

### 3.6.1 DLL Philosophy

The Cheetah DLL provides a robust approach to allow present-day Cheetah-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 communicate SPI through a Cheetah device. At the time the program is built, the Cheetah software is released as version 1.2. The Cheetah 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 Cheetah 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 Cheetah 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).

### 3.6.2 DLL Location

Total Phase provides language bindings that can be integrated into any custom application. The default behavior of locating the Cheetah 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 `CH_UNABLE_TO_LOAD_LIBRARY` error will be returned by the binding function.

### 3.6.3 DLL Versioning

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

Top

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

### 3.7.1 Overview

The Cheetah Rosetta language bindings make integration of the Cheetah API into custom applications simple. Accessing Cheetah functionality simply requires function calls to the Cheetah 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 `cheetah.h` file included with the API software package in any C or C++ source module. The module may now use any Cheetah API call listed in `cheetah.h`.
2. Compile and link `cheetah.c` with your application. Ensure that the include path for compilation also lists the directory in which `cheetah.h` is located if the two files are not placed in the same directory.
3. Place the Cheetah 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.

#### 3.7.2 Versioning

Since a new Cheetah 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., `cheetah.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, `CH_INCOMPATIBLE_LIBRARY`.

#### 3.7.3 Customizations

While the 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 `cheetah.c` for more details.

Top

### 3.8 Application Notes

#### 3.8.1 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.

#### 3.8.2 USB Scheduling Delays

More specifically, each API call that is used to send data to and from the Cheetah adapter can incur up to 125 µs (1 ms) in delay on the PC host for USB High Speed (Full Speed) connections. This is caused by the inherent design of the USB architecture. The operating system will queue any outgoing USB transfer request on the host until the next USB frame period. The frame period is 125 µs (1 ms). Thus, if the application attempts to execute several transactions in rapid sequence there can be 1-2 ms delay between each transaction plus any additional process scheduling delays introduced by the operating system. The best throughput can be achieved for single transactions that transfer a large number of bytes at a time.

## 4 Firmware

Top

### 4.1 Philosophy

The firmware included with the Cheetah adapter 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 Cheetah software automatically detects firmware compatibility and will inform the user if an upgrade is required.

Top

### 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

Top

## 5.1 Introduction

The API documentation describes the Cheetah Rosetta C bindings.

Top

## 5.2 General Data Types

The following definitions are provided for convenience.

```
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;
```

Top

## 5.3 Notes on Status Codes

Most of the Cheetah API functions 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 Cheetah DLL, `CH_UNABLE_TO_LOAD_LIBRARY` and `CH_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 Cheetah software. Furthermore, all API calls can potentially return the errors `CH_UNABLE_TO_LOAD_DRIVER` or `CH_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 (e.g., `CH_HEADER_VERSION` found in `cheetah.h` and `CH_CFILE_VERSION` found in `cheetah.c` ) to verify that there are no mismatches. Next, ensure that the Rosetta language binding (e.g., `cheetah.c` and `cheetah.h` ) are from the same release as the Cheetah 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 Cheetah handle can potentially return the error code `CH_INVALID_HANDLE` if the handle does not correspond to a valid Cheetah device that has already been opened. If this error is received, check the application code to ensure that the `ch_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 Cheetah device can also return the error `CH_COMMUNICATION_ERROR`. This means that while the Cheetah 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.

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.

Top

## 5.4 General

### 5.4.1 Interface

Top

#### Find Devices (ch_find_devices)

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

Get a list of ports to which Cheetah 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 `CH_PORT_NOT_FREE` ( `0x8000` ). Under Linux, such devices correspond to Cheetah adapters that are currently in use. Under Windows, such devices are currently in use, but it is not known if the device is a Cheetah adapter.

Example:

Devices are attached to port 0, 1, 2.
Ports 0 and 2 are available, and port 1 is in-use.
`devices = { 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 (ch_find_devices_ext)

Top

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

Get a list of ports and unique IDs to which Cheetah 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 size.

**Specific Error Codes**

None.

**Details**

This function is the same as `ch_find_devices()` except that it also returns the unique IDs of each Cheetah 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 Cheetah device (ch_open)

Top

```
Cheetah ch_open (int port_number);
```

Open a connection to the Cheetah at the given port.

**Arguments**

| | |
|---|---|
| port_number | The Cheetah port number. This port number is the the same as the one obtained from the `ch_find_devices function`. It is a zero-based number. |

**Return Value**

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

**Specific Error Codes**

| | |
|---|---|
| CH_UNABLE_TO_OPEN | The specified port is not connected to a Cheetah device or the port is already in use. |
| CH_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 ch_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 `ch_open_ext()` is recommended.

## Open a Cheetah device (ch_open_ext)

Top

```
Cheetah ch_open_ext (int port_number, CheetahExt *ch_ext);
```

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

**Arguments**

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

**Return Value**

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

**Specific Error Codes**

| | |
|---|---|
| CH_UNABLE_TO_OPEN | The specified port is not connected to a Cheetah device or the port is already in use. |
| CH_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 `ch_ext`. |

**Details**

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

The `CheetahExt` structure is described below:

```
struct CheetahExt { CheetahVersion version; /* Features of this device.
*/ int features; }
```

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

```
struct CheetahVersion { /* Software 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 hardware revision. The lower 16 bits gives * the
minimum accepted hardware revision. */ u32 hw_revs_for_sw; /* * Firmware
revisions that are compatible with this * software version. The top 16
bits gives the maximum * accepted firmware revision. The lower 16 bits
gives * the minimum accepted firmware 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 interface 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 `ch_open()` is recommended.

### Close a Cheetah (ch_close)

Top

```
int ch_close (Cheetah cheetah);
```

Close the connection to the given Cheetah.

#### Arguments

| | |
|---|---|
| cheetah | handle of a Cheetah adapter to be closed |

#### Return Value

The number of adapters 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 Cheetah adapters. The total number of Cheetah adapters closed is returned by the function.

### Get Port (ch_port)

Top

```
int ch_port (Cheetah cheetah);
```

Return the port number for this Cheetah handle.

#### Arguments

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

#### 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 (ch_unique_id)

Top

```
u32 ch_unique_id (Cheetah cheetah);
```

Return the unique ID of the given Cheetah device.

#### Arguments

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

#### Return Value

This function returns the unique ID for this Cheetah adapter. 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 (ch_status_string)

Top

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

Return the status string for the given status code.

**Arguments**

| | |
|---|---|
| status | status code returned by a Cheetah 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 (ch_version)

Top

```
int ch_version (Cheetah cheetah, CheetahVersion *version);
```

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

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| version | pointer to pre-allocated structure |

**Return Value**

A Cheetah status code is returned with `CH_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 `ch_open_ext` for the definition of `CheetahVersion`.

## Sleep (ch_sleep_ms)

Top

```
u32 ch_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 (ch_target_power)

Top

```
int ch_target_power (Cheetah cheetah, u08 power_mask);
```

Activate/deactivate target power pins 4 and 6.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| power_mask | enumerated values specifying power pin state. See Table 2. |

**Table 2** : `power_mask` enumerated types

| | |
|---|---|
| `CH_TARGET_POWER_OFF` | Disable target power pins |
| `CH_TARGET_POWER_ON` | Enable target power pins |
| `CH_TARGET_POWER_QUERY` | Queries the target power pin state |

**Return Value**

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

**Specific Error Codes**

None.

**Details**

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

### Host Interface Speed (ch_host_ifce_speed)

Top

```
int ch_host_ifce_speed (Cheetah cheetah);
```

Return the interface host speed.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

**Return Value**

This function returns enumerated values specifying the USB speed at which the host computer is communicating with the given Cheetah device. See Table 3.

**Table 3** : Host interface speed enumerated types

| | |
|---|---|
| `CH_HOST_IFCE_FULL_SPEED` | Full speed (12 Mbps) interface |
| `CH_HOST_IFCE_HIGH_SPEED` | High Speed (480 Mbps) interface |

**Specific Error Codes**

None.

**Details**

Used to determine the USB communication rate between the Cheetah device and the host. A High Speed USB interface is highly recommended to take full advantage of the high speed SPI interface.

Top

## 5.5 SPI Interface

### 5.5.1 SPI Overview

In order to use the Cheetah API to send SPI data across the bus at high speed, commands are accumulated in a queue until a call is made to batch shift all of the queued commands. The sequence of commands in the command queue for a typical SPI transaction is:

1. Call `ch_spi_queue_clear` to clear the command queue.

2. Call `ch_spi_queue_oe` to add a command to the queue to enable the Cheetah devices outputs on the SPI bus.
3. Call `ch_spi_queue_ss` to add a command to the queue to enable the slave select signal.
4. Call `ch_spi_queue_byte` and `ch_spi_queue_array` to queue data to be sent across the SPI bus.
5. Call `ch_spi_queue_ss` to queue a command to disable the slave select signal.
6. Optionally, call `ch_spi_queue_oe` to queue a command to disable the outputs of the Cheetah device.
7. Call `ch_spi_batch_shift` to send the accumulated accumulated commands across the SPI bus.

Please note that the queue is not cleared after it has been executed. This allows the same queue to be executed multiple times without re-queuing commands. The only way to clear a queue is by calling `ch_spi_queue_clear`.

If the user wishes to repeat the exact same shift sequence on the bus, the last step can be executed without the preceding ones. Also, the OE enable commands and SS assert/deassert commands are held persistent on the Cheetah device. For example, if the SS line is left asserted after one transaction, it will stay asserted until modified in a subsequent transaction.

The data sent from the slave device to the Cheetah device across the MISO line will be filled into the `data_in` parameter of the `ch_spi_batch_shift` function. Also, `ch_spi_batch_length` can be called prior to calling `ch_spi_batch_shift` to find out how large of an array needs to be allocated for the `data_in` argument.

The SPI master and slave must both be configured to use the same bit protocol (mode). Prior to sending any data across the SPI bus, the application should call `ch_spi_configure` and `ch_spi_bitrate` to set the proper configuration for the SPI communication.

This architecture for the Cheetah API allows for maximum flexibility of the control lines (SS1, SS2, and SS3) and provides for a mechanism to avoid expensive USB scheduling delays (as described in a previous section of this datasheet). For example, one can send several separate SPI packets with the following sequence:

1. Assert the OE.
2. Assert the appropriate SS, queue the data, and deassert SS.
3. Add optional delay on the bus.
4. Assert the appropriate SS, queue the data, and deassert SS.
5. Repeat the last two steps as many times as desired.
6. Call the batch shift function.

This grouping of packets into a single transactions can be very efficient for programming devices such as EEPROMs or flash memories that require a separate SPI packets for each page that needs to be written. The delay step would then correspond to the page write time specified in the memory's datasheet.

Note that the `ch_spi_batch_shift` function will only return to the caller after the entire queue has been shifted on the SPI bus. There is an alternative, non-blocking implementation that can be used in lieu of `ch_spi_batch_shift`. The functions `ch_spi_async_submit` and `ch_spi_async_collect` allow the user to submit a batch queue and perform other tasks (including assembling and submitting a subsequent queue) before collecting the results of the initial SPI shift operation. This is an advanced feature that has traditionally only been needed by a few users; a detailed description is provided in the relevant API documentation. The error code `CH_SPI_ASYNC_PENDING` can be returned by other synchronous functions in the event that asynchronous batches are pending (i.e., have not been yet collected).

### 5.5.2 SPI Functions

Top

#### Set Bitrate (ch_spi_bitrate)

```
int ch_spi_bitrate (Cheetah cheetah, int bitrate_khz);
```

Set the SPI bitrate in kilohertz.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| bitrate_khz | the requested bitrate in kHz. |

**Return Value**

This function returns the actual bitrate set.

**Specific Error Codes**

None.

**Details**

The power-on default bitrate is 1 MHz.

Only certain discrete bitrates are supported by the Cheetah adapter. As such, this actual bitrate set will be less than or equal to the requested bitrate unless the requested value is less than 100 kHz, in which case the Cheetah adapter will default to 100 kHz. The maximum supported speed for the Cheetah adapter is 40 MHz. Overclocking to 50 MHz is possible, but may not operate correctly.

If `bitrate_khz` is 0, the function will set the bit rate to the minimum value, 100 kHz.

## Configure SPI Bus Parameters (ch_spi_configure)

Top

```
int ch_spi_configure (Cheetah cheetah, CheetahSpiPolarity polarity,
CheetahSpiPhase phase, CheetahSpiBitorder bitorder, u08 ss_polarity);
```

Configure the SPI interface.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| polarity | the polarity of the clock signal (see Table 4 ) |
| phase | the phase of the clock signal to sample on (see Table 5 ) |
| bitorder | the bit order, big endian or little endian (see Table 6 ) |
| ss_polarity | the polarity of the slave select signals |

**Table 4** : `polarity` enumerated types

| | |
|---|---|
| `CH_SPI_POL_RISING_FALLING` | Clock idle low, active high |
| `CH_SPI_POL_FALLING_RISING` | Clock idle high, active low |

**Table 5** : `phase` enumerated types

| | |
|---|---|
| `CH_SPI_PHASE_SAMPLE_SETUP` | Sample on the leading edge of the clock signal |
| `CH_SPI_PHASE_SETUP_SAMPLE` | Sample on the trailing edge of the clock signal |

**Table 6** : `bitorder` enumerated types

| | |
|---|---|
| `CH_SPI_BITORDER_MSB` | Big-endian bit order |
| `CH_SPI_BITORDER_LSB` | Little-endian bit order |

**Return Value**

A Cheetah status code is returned with `CH_OK` on success.

**Specific Error Codes**

None.

**Details**

These configuration parameters specify how to clock the bits that are sent and received on the Cheetah SPI interface.

The `polarity` option specifies which transition constitutes the leading edge and which transition is the falling edge. For example, `CH_SPI_POL_RISING_FALLING` would configure the SPI to idle the SCLK clock line low. The clock would then

transition low-to-high on the leading edge and high-to-low on the trailing edge.

The `phase` option determines whether to sample or setup on the leading edge of the clock signal. For example, `CH_SPI_PHASE_SAMPLE_SETUP` would configure the SPI to sample on the leading edge and setup on the trailing edge of the clock.

For example, mode 3 in the figure found in the "SPI Background" chapter would correspond to the pair ( `CH_SPI_POL_FALLING_RISING`, `CH_SPI_PHASE_SETUP_SAMPLE` ).

The `bitorder` option is used to indicate whether LSB or MSB is shifted first.

The `ss_polarity` option is a bit mask that indicates whether each SS line is active high or active low. For example, setting `ss_polarity` to `0x05` would mean that SS3 and SS1 are active high and SS2 is active low.

### Clear Batch Queue (ch_spi_queue_clear)

Top

```
int ch_spi_queue_clear (Cheetah cheetah);
```

Clear the batch queue.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

**Return Value**

A Cheetah status code is returned with `CH_OK` on success.

**Specific Error Codes**

None.

**Details**

All queued data and commands are removed from the queue.

### Queue the Cheetah Output Enable (ch_spi_queue_oe)

Top

```
int ch_spi_queue_oe (Cheetah cheetah, u08 oe);
```

Enable/disable the outputs.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| oe | 0 to disable the outputs, and 1 to enable |

**Return Value**

A Cheetah status code is returned with `CH_OK` on success.

**Specific Error Codes**

None.

**Details**

This function enables and disables the outputs on the Cheetah device. When enabled, the Cheetah device connects to the SPI bus and is ready to drive the signal lines. When the Cheetah output enable is disabled, the device disconnects from the bus and each signal line is held at their current values with very weak internal pull-up or pull-down on the Cheetah device.

Call this function to enable the Cheetah outputs before calling any of the other queue functions.

### Queue Slave Select Signals (ch_spi_queue_ss)

Top

```
int ch_spi_queue_ss (Cheetah cheetah, u08 active);
```

Queue Slave Select assertion/deassertion.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

| | |
|---|---|
| active | a bit mask based on the 3 SS lines where 1 corresponds to assertion and 0 to deassertion |

**Return Value**

A Cheetah status code is returned with `CH_OK` on success.

**Specific Error Codes**

None.

**Details**

This function adds a command to the batch queue to assert/deassert the SPI slave select lines. The `active` parameter is a bit mask. For example, setting `active` to `0x05` would mean that SS3 and SS1 are asserted and SS2 is deasserted. The polarity of the slave select is determined from a previous call to `ch_spi_configure`.

### Queue a Byte (ch_spi_queue_byte)

Top

```
int ch_spi_queue_byte (Cheetah cheetah, int count, u08 data_out);
```

Queue a byte value to be sent across the bus one or more times.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| count | number of bytes to put in queue |
| data_out | value of the byte to queue |

**Return Value**

Returns the actual number of times the byte in `data_out` was added to the queue. This should equal `count`.

**Specific Error Codes**

None.

**Details**

Queues `count` number of bytes to send and sets each byte to the value of `data_out`.

For this command, and the `ch_spi_queue_array` command, the polarity, phase, and bit ordering will be determined by the configuration set with the `ch_spi_configure` function.

### Queue a Byte Array (ch_spi_queue_array)

Top

```
int ch_spi_queue_array (Cheetah cheetah, u16 num_bytes, const u08 *
data_out);
```

Queue an array of bytes to be sent across the bus.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| num_bytes | number of bytes in `data_out` |
| data_out | byte array to queue |

**Return Value**

Returns the number of bytes queued. This should equal `num_bytes`.

**Specific Error Codes**

None.

**Details**

The array will be processed with a minimal run length encoding algorithm, so repeated sequences of single bytes will be sent more efficiently. This helps reduce outgoing (host to Cheetah) bandwidth on the USB bus.

For this command, and the `ch_spi_queue_byte` command, the polarity, phase, and bit ordering will be determined by the configuration set with the `ch_spi_configure` function.

### Queue a Delay in Cycles (ch_spi_queue_delay_cycles)

Top

```
int ch_spi_queue_delay_cycles (Cheetah cheetah, int cycles);
```

Queue a delay value on the bus in units of clock cycles.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| cycles | cycles of delay to add to the outbound shift |

**Return Value**

Returns the actual number of cycles of delay that were queued.

**Specific Error Codes**

None.

**Details**

Queues `cycles` amount of delay on the bus. These are in units of clock cycles as set with `ch_spi_bitrate`. The delays can only be queued in multiples of 8. The function will return the actual number of cycles queued. The requested number of cycles will be rounded up to the next multiple of 8.

The requested number of cycles must be greater than zero and less than or equal to $2^{32}$ -9. If the requested number of cycles is out of bounds, no delay is queued and the function will return 0.

### Queue a Delay in Nanoseconds (ch_spi_queue_delay_ns)

Top

```
int ch_spi_queue_delay_ns (Cheetah cheetah, int nanoseconds);
```

Queue a delay value on the bus in units of nanoseconds.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| nanoseconds | amount of time for delay in nanoseconds |

**Return Value**

Returns the actual number of nanoseconds of delay that were queued.

**Specific Error Codes**

None.

**Details**

Queues `nanoseconds` amount of delay on the bus. The fundamental unit of delay that can be queued on the Cheetah SPI bus is 8 times the clock period. Therefore, requested delay will be rounded up to the next even multiple of this time span. The function will return the actual number of nanoseconds queued.

The requested number of nanoseconds must be greater than zero and less than or equal to 2 seconds. If the requested number of nanoseconds is out of bounds, no delay is queued and the function will return 0.

### Get Shift Length (ch_spi_batch_length)

Top

```
int ch_spi_batch_length (Cheetah cheetah);
```

Returns the number of data bytes in the queue.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

**Return Value**

Returns the number of data bytes in the queue. This does not include the commands (SS assertion, OE, etc). It only corresponds to the number of bytes that will be shifted out on the SPI bus. The value returned is also the number of bytes to expect from the slave device when the currently queued commands are executed.

**Specific Error Codes**

None.

**Details**

None.

### Execute SPI Shift (ch_spi_batch_shift)

Top

```
int ch_spi_batch_shift (Cheetah cheetah, int num_bytes, u08 * data_in);
```

Perform the SPI shift operations in the queue.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| num_bytes | number of bytes to put into `data_in` |
| data_in | buffer to place data received from slave device |

**Return Value**

The number of bytes sent by the Cheetah device across the SPI bus.

**Specific Error Codes**

| | |
|---|---|
| CH_SPI_BATCH_EMPTY_QUEUE | The queue was empty. |

**Details**

This function performs all of the accumulated commands in the queue and shifts them in order onto the SPI bus. After the operation completes, the batch queue is not cleared. Therefore, this function may be called repeatedly if the same sequence of commands is to be shifted across the bus multiple times.

As data is shifted by the Cheetah device onto the MOSI line, the slave device will shift the same amount of data back across the MISO line. This function will put the first `num_bytes` number of bytes received from the slave device into the data_in array. To ensure that all of the data from the slave device is captured, call `ch_spi_batch_length` to determine how much data to expect from the slave device. Setting `num_bytes` to `0` is permissible for cases where the data from slave device is not required and can simply be discarded.

If the number of bytes received from the slave device is less than `num_bytes`, only the number of bytes received will be put into `data_in`. However, all of the outgoing bytes in the queue will still be shifted. When setting `num_bytes` to a value less than the total outgoing length, a special optimization will automatically take effect. This optimization will help reduce the traffic on the USB bus in the inbound direction (Cheetah to host). This optimization introduces an 8 clock cycle delay in the operation of the Cheetah device at the point that the Cheetah device stops sending the slave response back to the host.

### Submit an Asynchronous SPI Shift (ch_spi_async_submit)

Top

```
int ch_spi_async_submit (Cheetah cheetah);
```

Submit the SPI shift operations in the queue for asynchronous execution.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |

**Return Value**

The number of bytes to be sent by the Cheetah device across the SPI bus.

**Specific Error Codes**

| | |
|---|---|
| CH_SPI_BATCH_EMPTY_QUEUE | The queue was empty. |
| CH_SPI_ASYNC_MAX_REACHED | The maximum number of outstanding asynchronous batches has been reached. |

| | |
|---|---|
| CH_SPI_ASYNC_EXCESS_DELAY | The queue has a queued delay after the last shifted byte, which is not permissible for asynchronous batch shifting. |

**Details**

This function will submit the current batch queue asynchronously to the Cheetah. A temporary outgoing buffer will be created to store the batch queue. An internal incoming buffer will be also created to asynchronously capture the slave response data. The application programmer does not have to explicitly manage these two buffers. The function will immediately return after queuing this batch onto the USB, rather than waiting for the shift to complete on the SPI bus.

At this point, the application can submit another batch to the queue. This can be done immediately by submitting the same queue a second time without altering it the application simply needs to call `ch_spi_async_submit` again. Or, the application may clear the queue and assemble a different batch all together (see the ch_spi_queuefamily of functions). Any subsequent calls to `ch_spi_async_submit` will again create a temporary outgoing buffer and copy the current batch into it. Likewise, a temporary incoming buffer will also be created.

Note that the submitted batch should be sufficiently long (in real time) so that it does not complete before the application can submit more batches (and also collect the first batch). This will allow the adjacent batches to shift with very little delay between them. How long to be safe? First, there is always the possibility that the applications process could be scheduled out by the operating system before it has an opportunity to submit the subsequent batch. The operating system scheduler timeslice may be as much as 10ms. Therefore, submitted batches should be long enough to bridge one, if not two, time slices. Second, if the application is performing its own functions between the submission of two batches, the length of the batches should be long enough to accommodate the CPU time of those functions.

Keep in mind that there can significant memory overhead for each asynchronous batch:

1. Up to 4 times the size of the outgoing number of bytes. In the worst case, if there are no sequential repeats of data, the outgoing buffer is approximately twice the size of the number of bytes shifted out on the SPI bus (this doesnt count SS# assert/deassert commands or intermediate delays) and there is potentially another factor of two due to kernel/user mode memory allocation. So if the application shifts 10 KB out in one batch, the outgoing buffer overhead is approximately 40 KB.
2. 2 times the size of the incoming buffer for each batch.
3. Size of the `data_in` buffer supplied to the `ch_spi_async_collect` function.

Hence, it is important to not queue many megabytes of batches with the asynchronous interface. Additionally, only a fixed number of batches can be submitted and be left pending prior to collection. This number is fixed to 16.

Finally, the asynchronous interface is only useful if the outgoing data of any asynchronous batch does not rely on the return MISO data of a previous asynchronous batch.

## Collect an Asynchronous SPI Shift (ch_spi_async_collect)

Top

```
int ch_spi_async_collect (Cheetah cheetah, int num_bytes, u08 * data_in);
```

Collect a previously submitted asynchronous SPI shift queue.

**Arguments**

| | |
|---|---|
| cheetah | handle of a Cheetah adapter |
| num_bytes | number of bytes to put into `data_in` |
| data_in | buffer to place data received from slave device |

**Return Value**

The number of bytes sent by the Cheetah device across the SPI bus.

**Specific Error Codes**

| | |
|---|---|
| CH_SPI_ASYNC_EMPTY | There are no outstanding asynchronous batches to be collected. |

**Details**

This function can be called at anytime after submitting a batch for asynchronous processing. It will block until the first pending batch completes. For example, if there are 5 batches outstanding, this function will return after 1 batch has fully completed, leaving the other 4 batches still outstanding.

The function will fill the user-supplied buffer with the incoming MISO data for the completed batch. The incoming data will be copied from the internal incoming buffer originally created by `ch_spi_async_submit` into the user supplied

`data_in` buffer. The internal buffers (both outgoing and incoming) for the completed batch will be deleted before this function returns to the application.

If `ch_close` is called without collecting pending asynchronous batches, those batches will be canceled, even if they are in progress. All temporary buffers will be freed as well.

A suggested mode of operation would be as follows:

1. Submit 2-4 asynchronous batches, each with about 40 ms worth of SPI shifting. A single batch can contain many SPI commands / packets, of course.
2. Call `ch_spi_async_collect` to collect one batch. Signal another thread to process this data.
3. Submit another asynchronous batch to replace the one that was just collected.
4. Repeat steps 2 & 3 while your other thread simply processes the data.

Note that this merely is a recommendation for use and developers can modify this procedure as it suits their own application requirements.

The application must keep full accounting of how many batches have been submitted and how many are collected during each step of the process. It is even possible that the application will not need multiple threads if it can process the data between steps 2 and 3 and guarantee the algorithm will not take too long to process the incoming data.

Top

## 5.6 Error Codes

**Table 7** : Cheetah API Error Codes

| Literal Name | Value | ch_status_string() return value |
|---|---|---|
| CH_OK | 0 | ok |
| CH_UNABLE_TO_LOAD_LIBRARY | -1 | unable to load library |
| CH_UNABLE_TO_LOAD_DRIVER | -2 | unable to load USB driver |
| CH_UNABLE_TO_LOAD_FUNCTION | -3 | unable to load binding function |
| CH_INCOMPATIBLE_LIBRARY | -4 | incompatible library version |
| CH_INCOMPATIBLE_DEVICE | -5 | incompatible device version |
| CH_INCOMPATIBLE_DRIVER | -6 | incompatible driver version |
| CH_COMMUNICATION_ERROR | -7 | communication error |
| CH_UNABLE_TO_OPEN | -8 | unable to open device |
| CH_UNABLE_TO_CLOSE | -9 | unable to close device |
| CH_INVALID_HANDLE | -10 | invalid device handle |
| CH_CONFIG_ERROR | -11 | configuration error |
| CH_UNKNOWN_PROTOCOL | -12 | unknown protocol |
| CH_STILL_ACTIVE | -13 | still active |
| CH_FUNCTION_NOT_AVAILABLE | -14 | function not available |
| CH_OS_ERROR | -15 | os error |
| CH_SPI_WRITE_ERROR | -100 | spi write error |
| CH_SPI_BATCH_EMPTY_QUEUE | -101 | spi batch empty queue |
| CH_SPI_BATCH_SHORT_BUFFER | -102 | spi batch short buffer |
| CH_SPI_ASYNC_EMPTY | -103 | spi async empty |
| CH_SPI_ASYNC_PENDING | -104 | spi async pending |
| CH_SPI_ASYNC_MAX_REACHED | -105 | spi async max reached |
| CH_SPI_ASYNC_EXCESS_DELAY | -106 | spi async excess trailing delay |

# 6 Legal / Contact

Top

## 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.

Top

## 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.

Top

## 6.3 Contact Information

Total Phase can be found on the Internet at https://www.totalphase.com/. If you have support-related questions, please go to the Total Phase website. For sales inquiries, please contact sales@totalphase.com.

2006-2014 Total Phase, Inc.

All rights reserved.