Aardvark I²C/SPI Embedded Systems Interface

Features

- I²C – Two wire interface
  - Standard mode (100 kbps) and fast mode (400 kbps)
  - Variable bitrate (1-800 kbps)
  - Byte throughput rates are near theoretical maximums
  - Master transmit and receive
  - Asynchronous slave transmit and receive
  - Repeated start, 10-bit addressing, and combined format
- SPI – 4 wire serial communication protocol
  - Up to 8 Mbps master signaling rate
  - Up to 4 Mbps slave signaling rate
  - Full duplex master transmit/receive
  - Asynchronous slave transmit/receive
  - Configurable slave select polarity for master mode
- GPIO – General Purpose Input/Output
  - General purpose signaling on I2C and/or SPI pins
  - Internal caching of GPIO configuration
- I²C Monitor
  - Unobtrusively record traffic on an I2C bus
  - Supports bus speeds up to 125 kHz
- USB host communication
  - Up to 3 Mbps transfer to host PC
  - Powered from USB, eliminating power adapters
  - Reports as a full-speed device (12 Mbps) to USB 2.0 hosts
- Software API
  - Windows, Linux, and Mac OS X compatible
  - Easy to integrate application interface
  - Upgradeable Firmware
  - Field upgradeable over USB
  - Error checking prevents upgrade disasters

Summary

The Aardvark I²C/SPI Embedded Systems Interface is a host adapter through USB. It allows a developer to interface a host PC to a downstream embedded system environment and transfer serial messages using the I²C and SPI protocols. I²C and SPI functionality can be used concurrently. Additionally, the I²C and/or SPI pins can be used for general purpose signaling when the respective subsystem is not in use. An I²C monitoring feature is also available to unobtrusively record traffic on an I²C bus.

TOTAL PHASE

AARDVARK I²C/SPI

Aardvark I²C/SPI
Embedded Systems Interface

Data Sheet v5.11
September 15, 2009

1 General Overview

1.1 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).

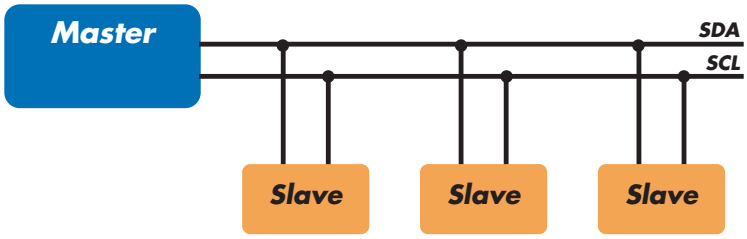

Figure 1: *Sample I²C Implementation.*
Regardless of how many slave units are attached to the I²C bus, there are only two signals connected to all of them. Consequently, there is additional overhead because an addressing mechanism is required for the master device to communicate with a specific slave 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 2 shows a sample bitstream of the I²C protocol.

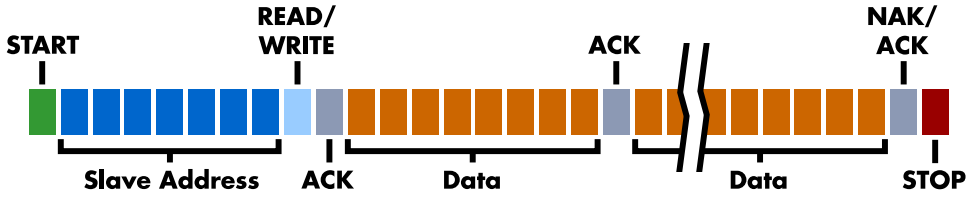

Figure 2: *I²C Protocol.*
Since there are only two wires, this protocol includes the extra overhead of the addressing and acknowledgement mechanisms.

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*

1.2 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).

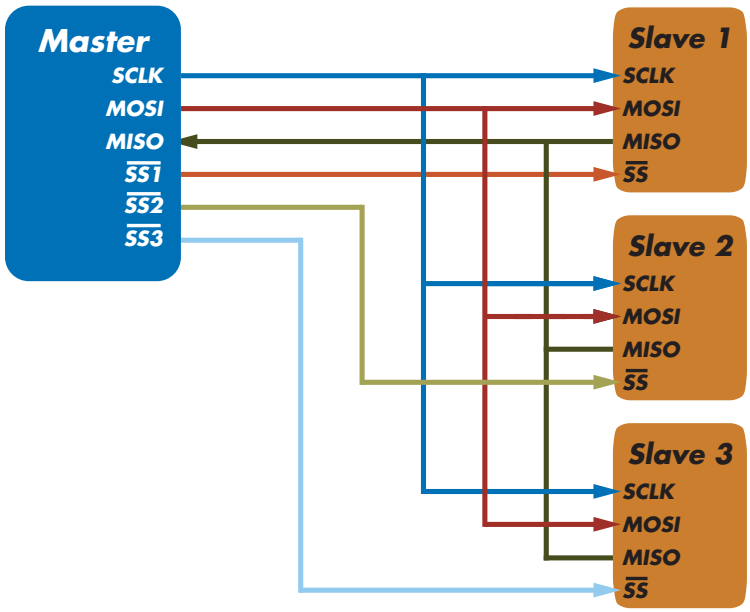

Figure 3: *Sample SPI Implementation.*
Each slave device requires a separate slave select signal (SS). This means that as devices are added, the circuit increases in complexity.

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 4.

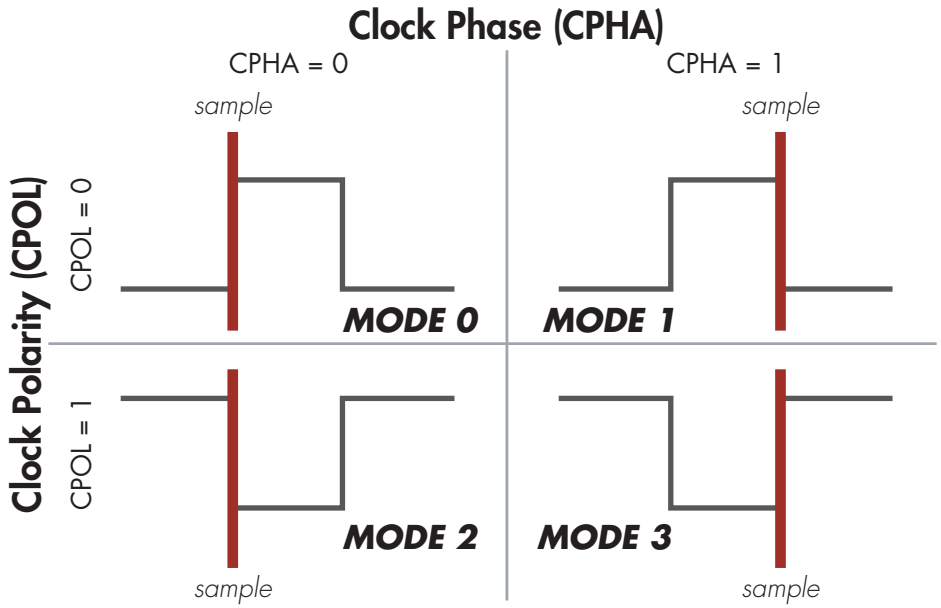

Figure 4: *SPI Modes*
The frame of the data exchange is described by two parameters, the clock polarity (CPOL) and the clock phase (CPHA). This diagram shows the four possible states for these parameters and the corresponding mode in SPI.

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 Pinouts

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 Aardvark adapter in the upright position (figure 5), pin 1 is in the top left corner and pin 10 is in the bottom right corner.

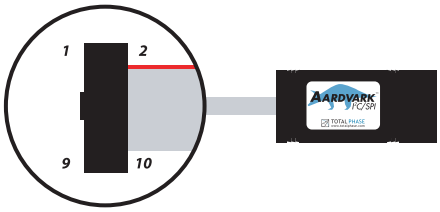

Figure 5: *The Aardvark I2C/SPI Host Adapter in the upright position.*
Pin 1 is located in the upper left corner of the connector and Pin 10 is located in the lower right corner of the connector.

If you flip your Aardvark adapter over (figure 6) 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.

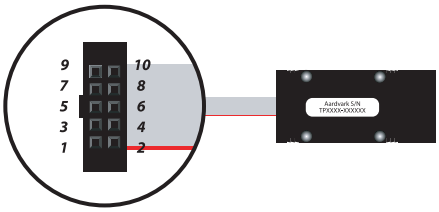

Figure 6: *The Aardvark I2C/SPI Host Adapter in the upside down position.*
Pin 1 is located in the lower left corner of the connector and Pin 10 is located in the upper right corner of the connector.

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 Aardvark adapter be on a common ground with the target system. If the grounds are not connected, the signaling is entirely unpredictable and communication will likely be corrupted. Two pins are connected to provide a solid ground path.

I²C Pins

SCL (Pin 1):

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

SDA (Pin 3):

Serial Data line – the 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 I²C or SPI EEPROM with the Aardvark adapter's power (which is provided by the USB port). It is ideal if the downstream device does not consume more than 20–30 mA. The Aardvark adapter is compatible with USB hubs as well

as USB host controllers (see section below on USB compliance). USB hubs are technically only rated to provide 100 mA per USB device. If the Aardvark adapter is directly plugged into a USB host controller, it can theoretically draw up to 500 mA total, leaving approximately 400 mA for any downstream target. However, the Aardvark adapter always reports itself to the host as a low-power device (<100 mA). Therefore, drawing large amounts of current from the host is not advisable.

NC/+5V (Pin 4): I²C Power
NC/+5V (Pin 6): SPI Power

By default, these pins are left unconnected at the time of shipping. For hardware versions 2.00 and greater, these pins are switched through the software API. A maximum of 50 mA may be drawn from each pin.

For hardware versions prior to 2.00, power cannot be supplied unless the appropriate jumpers are connected inside the Aardvark unit. To connect VDD to pins 4 and 6, connect jumpers J301 and J302.

Opening your Aardvark unit will void any hardware warranty. Any modifications are at the user's own risk.

2.2 Signal Levels/Voltage Ratings

Logic High Levels

All signal levels are nominally 3.3 volts (+/- 10%) logic high. This allows the Aardvark adapter to be used with both TTL and CMOS logic level 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 Aardvark adapter has built-in electrostatic discharge protection to prevent damage to the unit from high voltage static electricity.

Input Current

All inputs are high-impedance so the input current is approximately 1 µA.

Drive Current

The Aardvark adapter can drive all output signals with a maximum of 10 mA current source or sink. It may be possible to exceed this for short periods of time, but drawing more than 20 mA may damage the I/O buffers.

2.3 I²C Signaling Characteristics

Speed

As of hardware version 3.00, the Aardvark I²C master can operate at a maximum bitrate of 800 kbps and supports many intermediate bitrates between 1 kHz and the maximum. The power-on default bitrate is for the I²C master unit is 100 kbps.

For slave functionality, the Aardvark adapter can operate at any rate less than the maximum bitrate of 800 kbps.

For hardware versions before 3.00, the maximum I²C master bitrate is 663 kbps and the maximum I²C slave bitrate is 595 kbps.

It is not possible to send bytes at a throughput of exactly 1/8 times the bitrate. The I²C protocol requires that 9 bits are sent for every 8 bits of data. There is also a finite time required to setup a byte transmission. In the development of the Aardvark adapter, many optimizations have been employed to decrease this setup time. This allows byte throughputs within each transaction to be very close to the theoretical maximum byte throughput of 1/9 the bitrate.

There can be extra overhead introduced by the operating system between calls to the Aardvark API. These delays will further reduce the overall throughput across multiple transactions. To achieve the fastest throughput, it is advisable to send as many bytes as possible in a single transaction (i.e., a single call to the Aardvark API).

Pull-up Resistors

There is a 2.2K resistor on each I²C line (SCL, SDA). The lines are effectively pulled up to 3.3V, resulting in approximately 1.5 mA of pull-up current. If the Aardvark adapter is connected to an I²C bus that also includes pull-up resisters, the total pull-up current could be potentially larger. The I²C specification allows for a maximum of 3 mA pull-up current on each I²C line.

A good rule of thumb is that if a downstream I²C device can sink more than 5 mA of current, the protocol should operate properly. Stronger pull-up resistors and larger sink currents may be required for fast bitrates, especially if there is a large amount of capacitance on the bus. The Aardvark device is able to sink approximately 10 mA, so it is possible to have two Aardvark devices communicate with each other as master and slave, with both devices' pull-up resistors enabled.

Hardware versions 2.00 and later have the ability to switch the pull-up resistors by through the software API. Refer to the API section for more details.

I²C Clock Stretching

When the Aardvark adapter is configured as an I²C master, it supports both inter-bit and inter-byte slave clock-stretching. If a slave device pulls SCL low during a transaction, the adapter will wait until SCL has been released before continuing with the transaction.

Known I²C Limitations

Since firmware version 2.00, the Aardvark adapter supports the repeated start, 10-bit addressing, and combined format features of the I²C protocol.

Since firmware version 3.00, the Aardvark adapter can keep the slave functions enabled even while master operations are executed through the same adapter. The Aardvark adapter can even respond to slave requests immediately after losing bus arbitration during the slave addressing phase of a master transaction.

Multi-master is also supported with the following features:

1. If there is a bus collision during data transmission and the Aardvark adapter loses the bus, the transaction will be be cut short and the host API will report that fewer bytes were transmitted than the requested number. This condition can be distinguished from the case in which the downstream slave cuts short the transmission by sending a NACK by using the function `aa_i2c_read_ext`.
2. Collisions are not detected in the following situations:
   - Between a REPEATED START condition and a data bit
   - Between a STOP condition and a data bit
   - Between a REPEATED START and a STOP condition

This constraint can be phrased in a different manner. Say that I²C master device **A** has a packet length of **X** bytes. If there is a second I²C master device, **B**, that sends packets of length greater than **X** bytes, the first **X** bytes should never contain exactly the same data as the data sent by device **A**. Otherwise the results of the arbitration will be undefined.

This is a constraint found with most I²C master devices used in a multi-master environment.

2.4 SPI Signaling Characteristics

SPI Waveforms

The SPI signaling is characterized by the waveforms in Figures 7 and 8.

Table 1: *SPI Timing Parameters*

Symbol	Parameter	Min	Max	Units
t_1	SS# assertion to first clock	10	20	µs
t_2	Last clock to SS# deassertion	5	10	µs
t_p	Clock period	125	8000	ns
t_d	Setup time (Master)	7	9	µs
t_d	Setup time (Slave)	4	n/a	µs
t_b	Time between start of bytes (Slave)	10	n/a	µs

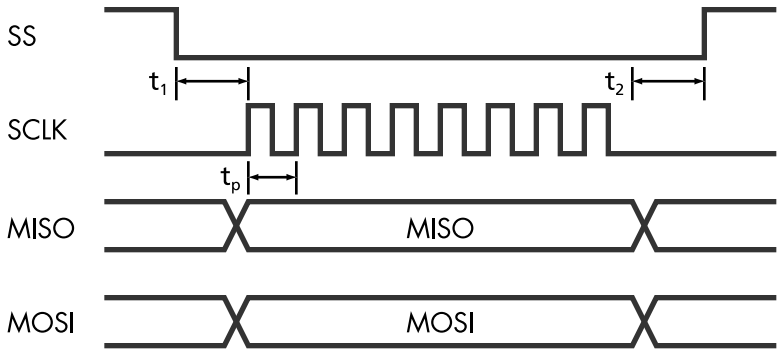

Figure 7: *SPI Waveform*
Three different times are of note: t_1, time from the assertion of SS# to the first clock cycle; t_2, time from the last clock cycle and the deassertion of SS#; t_p, clock period.

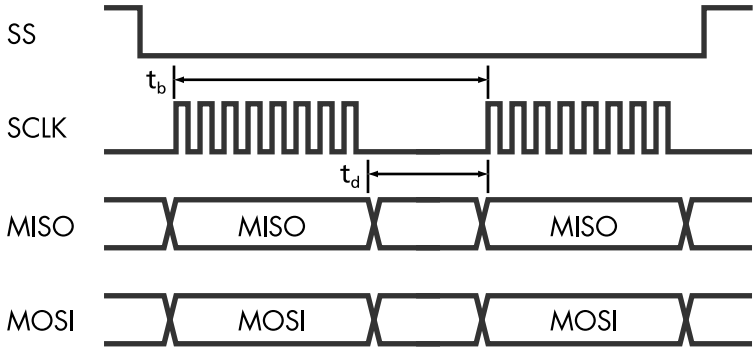

Figure 8: *SPI Byte Timing*
t_d is the setup time between SPI bytes and t_b is the total byte-to-byte time.

Speeds

The Aardvark SPI master can operate at bitrates of 125 kHz, 250 kHz, 500 kHz, 1 MHz, 2 MHz, 4 MHz, and 8 MHz. The power-on default bitrate is 1 MHz. The quoted bitrates are only achievable within each individual byte and does not extend across bytes. There is approximately 9 µs of setup time required in between each byte which results in a total transmission period of the byte transmission time plus 9 µs.

The Aardvark SPI slave can operate at any bitrate up to 4 Mbps. However, for correct transmission from slave to master there must be at least 4 µs delay between the end of byte **n** and start of byte **n+1** for the MISO data to be setup properly for byte **n+1**. Otherwise, the Aardvark adapter will simply return the data it just received. Likewise, for correct reception of data by the slave, there must be at least 10 µs between the start of byte **n** and the start of byte **n+1**.

Similarly, when the Aardvark adapter is configured to act as an SPI slave, and the slave select is pulled high to indicate the end of a transaction, there is a data processing overhead of sending the transaction to the PC host. As such, if the SPI master sends a subsequent transaction in rapid succession to the Aardvark slave, the data received by the Aardvark slave may be corrupted. There is no precise value for this minimum inter-transaction time, but a suggested spacing is approximately 100–200 µs.

Pin Driving

When the SPI interface is activated as a master, the slave select line (SS) is actively driven low. The MOSI and SCK lines are driven as appropriate for the SPI mode. After each transmission is complete, these lines are returned to a high impedance state. This feature allows the Aardvark adapter, following a transaction as a master SPI device, to be then reconnected to another SPI environment as a slave. The Aardvark adapter will not fight the master lines in the new environment.

Consequently, any SPI slave target to which the Aardvark adapter is interfaced must have a pull-up resistor on its slave select line, preventing fluttering of the voltage when the Aardvark adapter stops driving the signal. It is also advisable that every slave also have passive pull-ups on the MOSI and SCK lines. These pull-up resistors can be relatively weak — 100k should be adequate.

As a slave, the MOSI, SCK, and SS lines are configured as an input and the MISO line is configured as an output. This configuration is held as long as the slave mode is enabled (see the API documentation later in the datasheet).

Known SPI Limitations

There is currently an issue with SPI mode 2 (clock idles high, and sampling of MOSI is on the leading edge) that induces periodic bit corruptions in the most significant bit of certain bytes. The bug has been noted for SPI slave devices and there may also be corruptions when using this mode for sending or receiving messages as an SPI master. Unfortunately there is no fix for this problem and the best solution is to use another mode. If you have any questions regarding this issue please contact Total Phase support.

It is only possible to reliably send and receive transactions of 4 KiB or less as an SPI master or slave. This is due to operating system issues and the full-duplex nature of the SPI signaling.

Aardvark Device Power Consumption

The Aardvark adapter consumes less than 100 mA of current. This is within the USB 1.1 current specification for devices powered from a USB host adapter (500 mA maximum per device) or a USB hub (100 mA maximum per device).

2.5 USB 1.1 Compliance

The Aardvark adapter is USB 1.1 compliant and will also operate as a full speed (12 Mbps) device on a USB 2.0 hub or host controller

2.6 Temperature Specifications

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

3 Software

3.1 Compatibility

Overview

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

Windows Compatibility

The Aardvark software is compatible with Windows 2000 (SP4 or later), Windows XP (SP2 or later, 32-bit and 64-bit), and Windows Vista (32-bit and 64-bit). Legacy 16-bit Windows 95/98/ME operating systems are not supported.

Linux Compatibility

The Aardvark software is compatible with all standard 32-bit and 64-bit distributions of Linux with integrated USB support. Kernel 2.6 is required and the appropriate 32-bit system libraries are also required.

Mac OS X Compatibility

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

3.2 Windows USB Driver

Driver Installation

As of version 3.0, the Aardvark communications layer under Windows no longer uses a virtual serial port driver. The switch to a more direct USB driver should improve the installation and performance of PC and Aardvark adapter communication.

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 Aardvark adapter product page on the Total Phase website.

After the driver has been installed, plugging in an Aardvark 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 an Aardvark adapter is plugged into a system for the first time:

Windows 2000:

1. The Found New Hardware dialog window will appear during installation and will disappear when the installation completes.

Windows XP:

1. The Found New Hardware notification bubble will pop up from the system tray and state that the "Total Phase Aardvark I2C/SPI Host Adapter" has been detected.
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:

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 2000/XP or select "Control Panel | Hardware and Sound | Device Manager" for Windows Vista. The Aardvark device should appear under the "Universal Serial Bus Controllers" section.

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. It is critical that all Total Phase devices have been removed from your system before removing the USB drivers.

Windows 2000/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:

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

3.3 Linux USB Driver

As of version 3.0, the Aardvark communications layer under Linux no longer requires a specific kernel driver to operate. However, the user must ensure independently that the libusb library is installed on the system since the Aardvark library is dynamically linked to libusb.

Most modern Linux distributions use the udev subsystem to help manipulate the permissions of various system devices. This is the preferred way to support access to the Aardvark 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 Aardvark 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.

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 Aardvark 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 Aardvark adapter(s)

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: `aardvark` and `aardvark.usermap`. Please follow the following steps to enable hotplugging.

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

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 Aardvark

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

3.4 Mac OS X USB Driver

The Aardvark 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.

3.5 USB Port Assignment

The Aardvark 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 an Aardvark adapter is subsequently removed from the system, the remaining adapters shift their port numbers accordingly. With **n** Aardvark adapters attached, the allocated ports will be numbered from **0** to **n–1**.

Detecting Ports

To determine the ports to which the Aardvark adapters have been assigned, use the aa_find_devices routine as described in following API documentation.

3.6 Aardvark Dynamically Linked Library

DLL Philosophy

The Aardvark DLL provides a robust approach to allow present-day Aardvark-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 I²C or SPI through an Aardvark device. At the time the program is built, the Aardvark software is released

as version 1.2. The Aardvark 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 Aardvark 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 Aardvark 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 and Mac OS X, 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 Aardvark 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 application's current directory.
3. 32-bit system directory. Examples:
   - `c:\Windows\System32` [Windows 2000/XP/Vista 32-bit]
   - `c:\Windows\SysWow64` [Windows Vista 64-bit]
4. The windows directory. (Ex: `c:\Windows`)
5. 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 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 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 application's 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, an error will be returned by the binding function. The error code is AA_UNABLE_TO_LOAD_LIBRARY.

DLL Versioning

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

3.7 Rosetta Language Bindings: API Integration into Custom Applications

Overview

The Aardvark Rosetta language bindings make integration of the Aardvark API into custom applications simple. Accessing Aardvark functionality simply requires function calls to the Aardvark 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 aardvark.h file included with the API software package in any C or C++ source module. The module may now use any Aardvark API call listed in aardvark.h.

2. Compile and link `aardvark.c` with your application. Ensure that the include path for compilation also lists the directory in which `aardvark.h` is located if the two files are not placed in the same directory.

3. Place the Aardvark 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 Aardvark 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., `aardvark.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, `AA_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 `aardvark.c` for more details.

3.8 Application Notes

Asynchronous Messages

There is buffering within the Aardvark DLL, on a per-device basis, to help capture asynchronous messages. Take the case of the Aardvark adapter receiving I²C messages asynchronously. If the application calls the function to change the SPI bitrate while some unprocessed asynchronous messages are pending, the Aardvark adapter will transact the bitrate change but also save any pending I²C messages internally. The messages will be held until the appropriate API function is called.

The size of the I²C and SPI buffers are 16 KiB each and they can hold many separate transactions. The buffers are only used when an Aardvark API call is invoked. The buffer size is adequate since the overall limitation for asynchronous messages is fundamentally determined by the operating system's internal buffer size. This concept is explained below.

Receive Saturation

The Aardvark adapter can be configured as an I²C or SPI slave. A slave can receive messages asynchronously with respect to the host PC software. Between calls to the Aardvark 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, bytes will be dropped.

An overflow can occur when the Aardvark device receives asynchronous messages faster than the rate that they are processed — the receive link is "saturated." This condition can affect other synchronous communication with the Aardvark adapter. For example, if the SPI slave is receiving many unserviced messages (messages left pending in the operating systems buffer), a subsequent call to change the bitrate of I²C could fail in the following manner.

1. The software sends a command to the Aardvark adapter to change the bitrate.
2. The Aardvark adapter responds that the bitrate was set to a given value.
3. The response is lost since the operating system's receive buffer was full.

The requested bitrate has most likely been set by the Aardvark device, but the response was lost. A similar problem can happen when one attempts to disable the very slave that is saturating the incoming receive buffer! The API function sends a command to disable the slave, but the acknowledgment from the Aardvark adapter is lost. The API call will treat this as a communication error, but if the slave was actually disabled, a subsequent call to disable the slave should complete without errors.

The receive saturation problem can be improved in two ways. The obvious solution is to reduce the amount of traffic that is sent by the master between calls to the Aardvark API. This will require the ability to reconfigure the offending master device. The other option is to more regularly poll the slave to obtain any pending asynchronous messages. Keep in mind that each call to capture pending asynchronous data can have a timeout of up to 500 ms. If there is other time critical code that must be executed simultaneously, it is best to use the asynchronous polling function found in the API which allows for variable timeout values.

Threading

The Aardvark 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 Aardvark functionality, each Aardvark API call can be wrapped with a thread-safe locking mechanism before and after invocation. A call to any Aardvark API function that communicates with the host synchronously will also fetch any pending asynchronous messages, buffering them for subsequent calls to the asynchronous (slave) receive functions.

USB Scheduling Delays

Each API call that is used to send data to and from the Aardvark adapter can incur up to 1 ms in delay on the PC host. 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 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

4.1 Field Upgrades

Upgrade Philosophy

The Aardvark adapter is designed so that its internal firmware can be upgraded by the user, thereby allowing the inclusion of any performance enhancements or critical fixes available after the purchase of the device. The upgrade procedure is performed via USB and has several error checking facilities to ensure that the Aardvark adapter is not rendered permanently unusable by a bad firmware update. In the worst case scenario, a corruption can cause the Aardvark adapter to be locked until a subsequent clean update is executed.

The upgrade utility is also compatible with older devices that use the legacy virtual serial port communications drivers (pre-3.00 firmware revisions). The older serial port driver must be installed on your operating system. When listing such devices, the upgrade utility will report these devices with port numbers starting at 128. The devices will also be marked as "serial" as opposed to the "direct" identifier. Upgrading the legacy firmware will cause the Aardvark unit to automatically switch to using the new communications driver interface. If the host PC does not have the appropriate driver installed, the operating system may prompt the user for the new driver upon completion of the firmware upgrade. Please refer to the section on USB driver installation above for more information on how to install the new driver.

Upgrade Procedure

Here is the simple procedure by which the Aardvark firmware is upgraded.

1. Download the latest firmware from the Total Phase website.
2. Unzip the downloaded file. It should contain the `aaflash` utility. This utility contains the necessary information to perform the entire firmware update.
3. Run the appropriate version of `aaflash`:
   - `aaflash-win32.exe` on Windows
   - `aaflash-linux` on Linux
   - `aaflash-darwin` on Mac OS X

   It will first display the firmware version contained in the utility along with the required hardware version to run this firmware version.
4. It will list all of the detected devices along with their current firmware and hardware versions.
5. Select a device to upgrade. **Note the firmware and hardware version of the device before proceeding.** If the selected device's hardware is not suitable to accept the new firmware, an error will be printed and the utility will be reinvoked.
6. If the chosen device is acceptable, the `aaflash` utility will update the device with the new firmware. The process should take about 3 seconds, with a progress bar displayed during the procedure.

7. The upgraded Aardvark adapter should now be usable by any Aardvark-enabled application.

8. In the event that there was a malfunction in the firmware update, the Aardvark adapter may not be recognizable by an Aardvark-enabled application. Try the update again, since the Aardvark adapter has most likely become locked due to a corruption in the upgrade process. If the update still does not take effect, it is best to revert back to the previous firmware. This can be done by running a previous version of `aaflash` that contains an earlier firmware version. Check the Total Phase website or the distribution CD that was included with your Aardvark adapter for previous versions of the firmware.

5 API Documentation

5.1 Introduction

The API documentation that follows is oriented toward the Aardvark Rosetta C bindings. The set of API functions and their functionality is identical regardless of which Rosetta language binding is utilized. The only differences will be found in the calling convention of the functions. For further information on such differences please refer to the documentation that accompanies each language bindings in the Aardvark software distribution.

5.2 General Data Types

The following definitions are provided for convenience. All Aardvark data types are unsigned.

```
typedef unsigned char  aa_u08;
typedef unsigned short aa_u16;
typedef unsigned int   aa_u32;
```

5.3 Notes on Status Codes

Most of the Aardvark 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 regard to the loading of the underlying Aardvark DLL, AA_UNABLE_TO_LOAD_LIBRARY and AA_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 Aardvark software. Furthermore, all API calls can potentially return the error AA_UNABLE_TO_LOAD_FUNCTION. If this error is encountered, there is likely a serious version incompatibility that was not caught by the automatic version checking system. Where appropriate, compare the language binding versions (e.g., AA_HEADER_VERSION found in aardvark.h and AA_CFILE_VERSION found in aardvark.c) to verify that there are no mismatches. Next, ensure that the Rosetta language binding (e.g., aardvark.c and aardvark.h) are from the same release as the Aardvark 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 an Aardvark handle can return the error AA_INVALID_HANDLE if the handle does not correspond to a valid Aardvark device that has already been opened. If this error is received, check the application code to ensure that the aa_open command returned a valid handle and that this handle is not corrupted before being passed to the offending API function.

Finally, any I²C or SPI API call that communicates with an Aardvark device can return the error AA_COMMUNICATION_ERROR. This means that while the Aardvark handle is valid and the communication channel is open, there was an error receiving the acknowledgment response from the Aardvark adapter. This can occur in situations where the incoming data stream has been saturated by asynchronously received messages — an outgoing message is sent to the Aardvark adapter, but the incoming acknowledgment is dropped by the operating system as a result

of the incoming USB receive buffer being full. The error signifies that it was not possible to guarantee that the connected Aardvark device has processed the host PC request, though it is likely that the requested action has been communicated to the Aardvark adapter and the response was simply lost. For example, if the slave functions are enabled and the incoming communication buffer is saturated, an API call to disable the slave may return `AA_COMMUNICATION_ERROR` even though the slave has actually been disabled.

If either the I²C or SPI subsystems have been disabled by aa_configure, all other API functions that interact with I²C or SPI will return `AA_I2C_NOT_ENABLED` or `AA_SPI_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 (aa_find_devices)

```
int aa_find_devices (int      nelem,
                     aa_u16 * devices);
```

Get a list of ports to which Aardvark 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 `AA_PORT_NOT_FREE` (0x8000).

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 (aa_find_devices_ext)

```
int aa_find_devices_ext (int      num_devices,
                         aa_u16 * devices,
                         int      num_ids,
                         aa_u32 * unique_ids);
```

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

Arguments

`num_devices`: maximum number of device port numbers to return

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

`num_ids`: maximum number of unique IDs to return

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

Return Value

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

Specific Error Codes

None.

Details

This function is the same as `aa_find_devices()` except that is also returns the unique IDs of each Aardvark 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 an Aardvark device (aa_open)

```
Aardvark aa_open (int port_number);
```

Open the Aardvark port.

Arguments

`port_number`: The port is the same as the one obtained from function `aa_find_devices`. It is a zero-based number.

Return Value

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

Specific Error Codes

`AA_UNABLE_TO_OPEN`: The specified port is not connected to an Aardvark device or the port is already in use.

`AA_INCOMPATIBLE_DEVICE`: There is a version mismatch between the DLL and the firmware. The DLL is not of a sufficient version for interoperability with the firmware version or vice versa. See aa_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 `aa_open_ext()` is recommended.

The open function also deactivates all slave functionality. An Aardvark device could have potentially been opened, enabled as a slave, and configured to send asynchronous responses to a third-party master. If the controlling application quits without calling `aa_close()`, the port is freed but the slave functions can still be active. The open function deactivates slave functionality to ensure that the new application has access to an Aardvark device that is in a known-state. Also the I²C bus is freed, in the event that it was held indefinitely from a previous `AA_I2C_NO_STOP` transaction.

Open an Aardvark device (aa_open_ext)

```
Aardvark aa_open_ext (int port_number, AardvarkExt *aa_ext);
```

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

Arguments

`port_number`: same as `aa_open`

`aa_ext`: pointer to pre-allocated structure for extended version information available on open

Return Value

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

Specific Error Codes

`AA_UNABLE_TO_OPEN`: The specified port is not connected to an Aardvark device or the port is already in use.

`AA_INCOMPATIBLE_DEVICE`: There is a version mismatch between the DLL and the firmware. The DLL is not of a sufficient version for interoperability with the firmware version or vice versa. The version information will be available in the memory pointed to by `aa_ext`.

Details

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

The `AardvarkExt` structure is described below:

```
struct AardvarkExt {
    AardvarkVersion  version;
    /* Features of this device. */
    int              features;
}
```

The features field denotes the capabilities of the Aardvark device. See the API function `aa_features` for more information.

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

```
struct AardvarkVersion {
    /* Software, firmware, and hardware versions. */
    aa_u16 software;
    aa_u16 firmware;
    aa_u16 hardware;

    /* FW requires that SW must be >= this version. */
    aa_u16 sw_req_by_fw;

    /* SW requires that FW must be >= this version. */
    aa_u16 fw_req_by_sw;

    /* API requires that SW must be >= this version. */
    aa_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 firmware 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 `aa_open()` is recommended.

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

Close an Aardvark (aa_close)

```
int aa_close (Aardvark aardvark);
```

Close the Aardvark port.

Arguments

`aardvark`: handle of an Aardvark 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

An Aardvark adapter could have potentially been opened, enabled as a slave, and configured to send and receive asynchronous responses to and from a third-party master. A call to `aa_close()` will deactivate all slave functionality. Also the I²C bus is freed, in the event that it was held indefinitely from a previous `AA_I2C_NO_STOP` transaction.

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

Get Port (aa_port)

```
int aa_port (Aardvark aardvark);
```

Return the port number for this Aardvark handle.

Arguments

`aardvark`: handle of an Aardvark 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 Features (aa_features)

```
int aa_features (Aardvark aardvark);
```

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

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

The features of the Aardvark device are returned. These are a bit-mask of the following values.

```
#define AA_FEATURE_SPI          (1<<0)
#define AA_FEATURE_I2C          (1<<1)
#define AA_FEATURE_GPIO         (1<<3)
#define AA_FEATURE_I2C_MONITOR  (1<<4)
```

Specific Error Codes

None.

Details

None.

Get Unique ID (aa_unique_id)

```
aa_u32 aa_unique_id (Aardvark aardvark);
```

Return the unique ID of the given Aardvark device.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

This function returns the unique ID for this Aardvark 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 (aa_status_string)

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

Return the status string for the given status code.

Arguments

`status`: status code returned by an Aardvark 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.

Logging (aa_log)

```
int aa_log (Aardvark aardvark, int level, int handle);
```

Enable logging to a file.

Arguments

`aardvark`: handle of an Aardvark adapter

`level`: the logging detail level as described below

`handle`: a file descriptor

Return Value

An Aardvark status code is returned with `AA_OK` on success.

Specific Error Codes

None.

Details

The handle must be standard file descriptor. In C, a file descriptor can be obtained by using the ANSI C function "open" or by using the function "fileno" on a `FILE*` stream. A `FILE*` stream obtained using `fopen` or can correspond to the common `stdout` or `stderr` — available when including `stdlib.h`.

The logging detail level can be one of the following options.

0 – none
1 – error
2 – warning
3 – info
4 – debug

Note that if the handle is invalid, the application can crash during a logging operation.

Due to inconsistencies arising from how Microsoft handles linkage to the C runtime library, logging to a file may not work in Windows. However, logging to `stdout` and `stderr` is still supported. As a convenience, the following two constants are defined and can be passed as the `handle` argument.

- `AA_LOG_STDOUT`
- `AA_LOG_STDERR`

Version (aa_version)

```
int aa_version (Aardvark aardvark, AardvarkVersion *version);
```

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

Arguments

`aardvark`: handle of an Aardvark adapter

`version`: pointer to pre-allocated structure

Return Value

An Aardvark status code is returned with `AA_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 `aa_open_ext` for the definition of `AardvarkVersion`.

Configure (aa_configure)

```
int aa_configure (Aardvark aardvark, AardvarkConfig config);
```

Activate/deactivate individual subsystems (I²C , SPI, GPIO).

Arguments

`aardvark`: handle of an Aardvark adapter

`config`: enumerated type specifying configuration. See Table 2

Table 2: `config` *enumerated types*

`AA_CONFIG_GPIO_ONLY`	Configure all pins as GPIO. Disable both I²C and SPI.
`AA_CONFIG_SPI_GPIO`	Configure I²C pins as GPIO. Enable SPI.
`AA_CONFIG_GPIO_I2C`	Configure SPI pins as GPIO. Enable I²C .
`AA_CONFIG_SPI_I2C`	Disable GPIO. Enable both I²C and SPI.
`AA_CONFIG_QUERY`	Queries existing configuration (does not modify).

Return Value

The current configuration on the Aardvark adapter will be returned. The configuration will be described by the same values in `AardvarkConfig`.

Specific Error Codes

`AA_CONFIG_ERROR`: The I²C or SPI subsystem is currently active and the new configuration requires the subsystem to be deactivated.

Details

If either the I²C or SPI subsystems have been disabled by this API call, all other API functions that interact with I²C or SPI will return `AA_CONFIG_ERROR`.

If configurations are switched, the subsystem specific parameters will be preserved. For example if the SPI bitrate is set to 500 kHz and the SPI system is disabled and then enabled, the bitrate will remain at 500 kHz. This also holds for other parameters such as the SPI mode, SPI slave response, I²C bitrate, I²C slave response, etc.

However, if a subsystem is shut off, it will be restarted in a quiescent mode. That is to say, the I²C slave function will not be reactivated after re-enabling the I²C subsystem, even if the I²C slave function was active before first disabling the I²C subsystem.

Note: Whenever the configure command is executed and GPIO lines are enabled, the GPIO lines will be momentarily switched to high-Z before their direction and pullup configurations are executed.

Target Power (aa_target_power)

```
int aa_target_power (Aardvark aardvark, aa_u08 power_mask);
```

Activate/deactivate target power pins 4 and 6.

Arguments

`aardvark`: handle of an Aardvark adapter

`power_mask`: enumerated values specifying power pin state. See Table 3.

Table 3: `power_mask` *enumerated types*

AA_TARGET_POWER_NONE	Disable target power pins
AA_TARGET_POWER_BOTH	Enable target power pins
AA_TARGET_POWER_QUERY	Queries the target power pin state

Return Value

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

Specific Error Codes

`AA_INCOMPATIBLE_DEVICE`: The hardware version is not compatible with this feature. Only hardware versions 2.00 or greater support 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.

Asynchronous Polling (aa_async_poll)

```
int aa_async_poll (Aardvark aardvark, int timeout);
```

Check if there is any asynchronous data pending from the Aardvark adapter.

Arguments

`aardvark`: handle of an Aardvark adapter

`timeout`: timeout in milliseconds

Return Value

A status code indicating which types of asynchronous messages are available for processing. See Table 4.

Table 4: *Status code enumerated types*

AA_ASYNC_NO_DATA	No asynchronous data is available.
AA_ASYNC_I2C_READ	I²C slave read data is available. Use aa_i2c_slave_read to get data.
AA_ASYNC_I2C_WRITE	I²C slave write stats are available. Use aa_i2c_slave_write_stats to get data.
AA_ASYNC_SPI	SPI slave read data is available. Use aa_spi_slave_read to get data.
AA_ASYNC_I2C_MONITOR	I²C monitor data is available. Use aa_i2c_monitor_read to get data.

These codes can be bitwise OR'ed together if there are multiple types of data available.

Specific Error Codes

None.

Details

Recall that, like all other Aardvark functions, this function is not thread-safe.

If the timeout value is negative, the function will block indefinitely until data arrives. If the timeout value is 0, the function will perform a non-blocking check for pending asynchronous data.

As described before, the Aardvark software contains asynchronous queues that can be filled during synchronous operations on the Aardvark adapter. If data is already in one or more asynchronous queues, it will immediately return with all of the types of asynchronous data that are currently available. Further data may be pending in the operating system's incoming receive buffer, but the function will not examine that data. Hence any pending data in the operating system's incoming buffer will not be reported to the user until the Aardvark's software queues have been fully serviced.

If there is no data already available, this function will check the operating system's receive buffer for the presence of asynchronous data. The function will block for the specified timeout. It will then only report the type of the very first data that has been received. The function will not examine the remainder of the operating system's receive buffer to see what other asynchronous messages are pending.

One can employ the following technique to guarantee that all pending asynchronous data have been captured during each service cycle:

1. Call the polling function with a specified timeout.
2. If the polling function indicates that there is data available, call the appropriate service function once for each type of data that is available.
3. Next, call the polling function with a 0 timeout.
4. Call the appropriate service function once for each type of data that is available.
5. Repeat steps 3 and 4 until the polling function reports that there is no data available.

Sleep (aa_sleep_ms)

```
u32 aa_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.

5.5 I²C Interface

I²C Notes

1. It is not necessary to set the bitrate for the Aardvark I²C slave.
2. An I²C master operation read or write operation can be transacted while leaving the I²C slave functionality enabled. In a multi-master situation it is possible for the Aardvark adapter to lose the bus during the slave addressing portion of the transaction. If the other master that wins the bus subsequently addresses this Aardvark adapter's slave address, the Aardvark adapter will respond appropriately to the request using its slave mode capabilities.
3. It is always advisable to set the slave response before first enabling the slave. This ensures that valid data is sent to any requesting master.
4. It is not possible to receive messages larger than approximately 4 KiB as a slave due to operating system limitations on the asynchronous incoming buffer. As such, one should not queue up more than 4 KiB of total slave data between calls to the Aardvark API.
5. Since firmware revision 2.00 it is possible for the Aardvark I²C master to employ some of the advanced features of I²C . This is accomplished by the `AardvarkI2cFlags` argument type that is included in the `aa_i2c_read` and `aa_i2c_write` argument lists. The options in Table 5 are available can be logically OR'ed together to combine them for one operation.

Table 5: *I²C Advanced Feature Options*

`AA_I2C_NO_FLAGS`	Request no options.
`AA_I2C_10_BIT_ADDR`	Request that the provided address is treated as a 10-bit address. The Aardvark I²C subsystem will follow the Philips I²C specification when transmitting the address.
`AA_I2C_COMBINED_FMT`	Request that the Philips combined format is followed during a I²C read operation. Please see the Philips specification for more details. This flag does not have any effect unless a master read operation is requested and the `AA_I2C_10_BIT_ADDR` is also set.
`AA_I2C_NO_STOP`	Request that no stop condition is issued on the I²C bus after the transaction completes. It is expected that the PC will follow up with a subsequent transaction at which point a repeated start will be issued on the bus. Eventually an I²C transaction must be issued without the "no stop" option so that a stop condition is issued and the bus is freed.

6. Since firmware revision 3.00 it is possible for the Aardvark I²C master to return an extended status code for master read and master write transactions. These codes are described in Table 6 and are returned by the `aa_i2c_read_ext` and `aa_i2c_write_ext` functions, as well as the analogous slave API functions.

Table 6: *I²C Extended Status Code*

AA_I2C_STATUS_BUS_ERROR	A bus error has occurred. Transaction was aborted.
AA_I2C_STATUS_SLA_ACK	Bus arbitration was lost during master transaction; another master on the bus has successfully addressed this Aardvark adapter's slave address. As a result, this Aardvark adapter has automatically switched to slave mode and is responding.
AA_I2C_STATUS_SLA_NACK	The Aardvark adapter failed to receive acknowledgment for the requested slave address during a master operation.
AA_I2C_STATUS_DATA_NACK	The last data byte in the transaction was not acknowledged by the slave.
AA_I2C_STATUS_ARB_LOST	Another master device on the bus was accessing the bus simultaneously with this Aardvark adapter. That device won arbitration of the bus as per the I²C specification.
AA_I2C_STATUS_BUS_LOCKED	An I²C packet is in progress, and the time since the last I²C event executed or received on the bus has exceeded the bus lock timeout. This is most likely due to the clock line of the bus being held low by some other device, or due to the data line held low such that a start condition cannot be executed by the Aardvark. The bus lock timeout can be configured using the `aa_i2c_bus_timeout` function. The Aardvark adapter resets its own I²C interface when a timeout is observed and no further action is taken on the bus.
AA_I2C_STATUS_LAST_DATA_ACK	When the Aardvark slave is configured with a fixed length transmit buffer, it will detach itself from the I2C bus after the buffer is fully transmitted. The Aardvark slave also expects that the last byte sent from this buffer is NACK'ed by the opposing master device. This status code is returned by the Aardvark slave (see "Slave Write Statistics" API) if the master device instead ACKs the last byte. The notification can be useful when debugging a third-party master device.

These codes can provide hints as to why an impartial transaction was executed by the Aardvark adapter. In the event that a bus error occurs while the Aardvark adapter is idle and enabled as a slave (but not currently receiving a message), the adapter will return

the bus error through the `aa_i2c_slave_read` function. The length of the message will be 0 bytes but the status code will reflect the bus error.

General I²C

I2C Pullup (aa_i2c_pullup)

```
int aa_i2c_pullup (Aardvark aardvark, aa_u08 pullup_mask);
```

Activate/deactivate I²C pull-up resistors on SCL and SDA.

Arguments

`aardvark`: handle of an Aardvark adapter

`pullup_mask`: enumerated values specifying pullup state. See Table 7.

Table 7: `pullup_mask` *enumerated types*

`AA_I2C_PULLUP_NONE`	Disable SCL/SDA pull-up resistors
`AA_I2C_PULLUP_BOTH`	Enable SCL/SDA pull-up resistors
`AA_I2C_PULLUP_QUERY`	Queries the pull-up resistor state

Return Value

The current state of the I²C pull-up resistors on the Aardvark adapter will be returned. The configuration will be described by the same values as in the table above.

Specific Error Codes

`AA_INCOMPATIBLE_DEVICE`: The hardware version is not compatible with this feature. Only hardware versions 2.00 or greater support switchable pull-up resistors pins.

Details

Both pull-up resistors are controlled together. Independent control is not supported. This function may be performed in any operation mode.

Free bus (aa_i2c_free_bus)

```
int aa_i2c_free_bus (Aardvark aardvark);
```

Free the Aardvark I²C subsystem from a held bus condition (e.g., "no stop").

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

`AA_I2C_ALREADY_FREE_BUS`: The bus was already free and no action was taken.

Details

If the Aardvark I²C subsystem had executed a master transaction and is holding the bus due to a previous `AA_I2C_NO_STOP` flag, this function will issue the stop command and free the bus. If the bus is already free, it will return the status code `AA_I2C_BUS_ALREADY_FREE`.

Similarly, if the Aardvark I²C subsystem was placed into slave mode and in the middle of a slave transaction, this command will disconnect the slave from the bus, flush the last transfer,

and re-enable the slave. Such a feature is useful if the Aardvark adapter was receiving bytes but then was forced to wait indefinitely on the bus because of the absence of the terminating stop command. After disabling the slave, any pending slave reception will be available to the host through the usual `aa_i2c_slave_write_stats` and `aa_i2c_slave_read` API calls.

The bus is always freed (i.e., a stop command is executed if necessary) and the slave functions are disabled at software opening and closing of the device.

Set Bus Lock Timeout (aa_i2c_bus_timeout)

```
int aa_i2c_bus_timeout (Aardvark aardvark, aa_u16 timeout_ms);
```

Set the I²C bus lock timeout in milliseconds.

Arguments

`aardvark`: handle of an Aardvark adapter

`timeout_ms`: the requested bus lock timeout in ms.

Return Value

This function returns the actual timeout set.

Specific Error Codes

None.

Details

The power-on default timeout is 200 ms. The minimum timeout value is 10 ms and the maximum is 450 ms. If a timeout value outside this range is passed to the API function, the timeout will be restricted. The exact timeout that is set can vary based on the resolution of the timer within the Aardvark adapter. The nominal timeout that was set is returned back by the API function.

If `timeout_ms` is 0, the function will return the bus lock timeout presently set on the Aardvark adapter and the bus lock timeout will be left unmodified.

If the bus is locked during the middle of any I²C transaction (master transmit, master receive, slave transmit, slave receive) the appropriate extended API function will return the status code `AA_I2C_STATUS_BUS_LOCKED` as described in the preceding "Notes" section. The bus lock timeout is measured between events on the I²C bus, where an event is a start condition, the completion of 9 bits of data transfer, a repeated start condition, or a stop condition. For example, if a full 9 bits are not completed within the bus lock timeout (due to clock stretching or some other error), the bus lock error will be triggered.

Please note that once the Aardvark adapter detects a bus lock timeout, it will abort its I²C interface, even if the timeout condition is seen in the middle of a byte. When the Aardvark is acting as an I²C mater device, this may result in only a partial byte being executed on the bus.

I²C Master

Set Bitrate (aa_i2c_bitrate)

```
int aa_i2c_bitrate (Aardvark aardvark, int bitrate_khz);
```

Set the I²C bitrate in kilohertz.

Arguments

`aardvark`: handle of an Aardvark 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 100 kHz.

Only certain discrete bitrates are supported by the Aardvark I²C master interface. As such, this actual bitrate set will be less than or equal to the requested bitrate.

If `bitrate_khz` is 0, the function will return the bitrate presently set on the Aardvark adapter and the bitrate will be left unmodified.

Master Read (aa_i2c_read)

```
int aa_i2c_read (Aardvark          aardvark,
                 aa_u16            slave_addr,
                 AardvarkI2cFlags  flags,
                 aa_u16            num_bytes,
                 aa_u08 *          data_in);
```

Read a stream of bytes from the I²C slave device.

Arguments

`aardvark`: handle of an Aardvark adapter

`slave_addr`: the slave from which to read

`flags`: special operations as described in "Notes" section and below

`num_bytes`: the number of bytes to read (maximum 65535)

`data_in`: pointer to data

Return Value

Number of bytes read.

Specific Error Codes

`AA_I2C_READ_ERROR`: There was an error reading from the Aardvark adapter. This is most likely a result of a communication error.

Details

For ordinary 7-bit addressing, the lower 7 bits of `slave_addr` should correspond to the slave address. The topmost bits are ignored. The Aardvark I²C subsystem will assemble the address along with the `R/W` bit after grabbing the bus. For 10-bit addressing, the lower 10 bits of `addr` should correspond to the slave address. The Aardvark adapter will then assemble the address into the proper format as described in the Philips specification, namely by first issuing an write transaction on the bus to specify the 10-bit slave and then a read transaction to read the requested number of bytes. The initial write transaction can be skipped if the "Combined Format" feature is requested in conjunction with the 10-bit addressing functionality.

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

It is possible to read zero bytes from the slave. In this case, num_bytes is set to 0 and the data argument is ignored (i.e., it can be 0 or point to invalid memory). However, due to the nature of the I²C protocol, it is not possible to address the slave and not request at least one byte. Therefore, one byte is actually received by the host, but is subsequently thrown away.

If the number of bytes read is zero, the following conditions are possible.

- The requested slave was not found.
- The requested slave is on the bus but refuses to acknowledge its address.
- The Aardvark adapter was unable to seize the bus due to the presence of another I²C master. **Here, the arbitration was lost during the slave addressing phase — results can be unpredictable**.
- Zero bytes were requested from a slave. The slave acknowledged its address and returned 1 byte. That byte was dropped.

Ordinarily the number of bytes read, if not 0, will equal the requested number of bytes. One special scenario in which this will not happen is if the Aardvark adapter loses the bus during the data transmission due to the presence of another I²C master.

If the slave has fewer bytes to transmit than the number requested by the master, the slave will simply stop transmitting and the master will receive 0xff for each remaining byte in the transmission. This behavior is in accordance with the I²C protocol.

Additionally, the flags argument can be used to specify a "sized read" operation. If the flag includes the value AA_I2C_SIZED_READ, the Aardvark adapter will treat the first byte received from the slave as a packet length field. This length denotes the number of bytes that the slave has available for reading (not including the length byte itself). The Aardvark adapter will continue to read the minimum of num_bytes-1 and the length field. The length value must be greater than 0. If it is equal to 0, it will be treated as though it is 1. In order to support protocols that include an optional checksum byte (e.g., SMBus) the flag can alternatively be set to AA_I2C_SIZED_READ_EXTRA1. In this case the Aardvark will read one more data byte beyond the number specified by the length field.

Master Read Extended (aa_i2c_read_ext)

```
int aa_i2c_read_ext (Aardvark          aardvark,
                     aa_u16            slave_addr,
                     AardvarkI2cFlags  flags,
                     aa_u16            num_bytes,
                     aa_u08 *          data_in,
                     aa_u16 *          num_read);
```

Read a stream of bytes from the I²C slave device with extended status information.

Arguments

aardvark: handle of an Aardvark adapter

slave_addr: the slave from which to read

flags: special operations as described previously

num_bytes: the number of bytes to read (maximum 65535)

data_in: pointer to data

num_read: the actual number of bytes read

Return Value

Status code (see "Notes" section).

Specific Error Codes

None.

Details

This function operates exactly like aa_i2c_read, except that the return value now specifies a status code. The number of bytes read is returned through an additional pointer argument at the tail of the parameter list.

The status code allows the user to discover specific events on the I²C bus that would otherwise be transparent given only the number of bytes transacted. The "Notes" section describes the status codes.

For a master read operation, the AA_I2C_STATUS_DATA_NACK flag is not used since the acknowledgment of data bytes is predetermined by the master and the I²C specification.

Master Write (aa_i2c_write)

```
int aa_i2c_write (Aardvark          aardvark,
                  aa_u16            slave_addr,
                  AardvarkI2cFlags  flags,
                  aa_u16            num_bytes,
                  const aa_u08 *    data_out);
```

Write a stream of bytes to the I²C slave device.

Arguments

aardvark: handle of an Aardvark adapter

slave_addr: the slave from which to read

flags: special operations as described in "Notes" section

num_bytes: the number of bytes to write (maximum 65535)

data_out: pointer to data

Return Value

Number of bytes written.

Specific Error Codes

AA_I2C_WRITE_ERROR: There was an error reading the acknowledgment from the Aardvark adapter. This is most likely a result of a communication error.

Details

For ordinary 7-bit addressing, the lower 7 bits of slave_addr should correspond to the slave address. The topmost bits are ignored. The Aardvark I²C subsystem will assemble the address along with the R/W bit after grabbing the bus. For 10-bit addressing, the lower 10 bits of addr should correspond to the slave address. The Aardvark adapter will then assemble the address

into the proper format as described in the Philips specification. There is a limitation that a maximum of only 65534 bytes can be written in a single transaction if the 10-bit addressing mode is used.

The `slave_addr` `0x00` has been reserved in the I²C protocol specification for general call addressing. I²C slaves that are enabled to respond to a general call will acknowledge this address. The general call is not treated specially in the Aardvark I²C master. The user of this API can manually assemble the first data byte if the hardware address programming feature with general call is required.

It is actually possible to write 0 bytes to the slave. The slave will be addressed and then the stop condition will be immediately transmitted by the Aardvark adapter. No bytes are sent to the slave, so the `data` argument is ignored (i.e., it can be 0 or point to invalid memory).

If the number of bytes written is zero, the following conditions are possible.

- The requested slave was not found.
- The requested slave is on the bus but refuses to acknowledge its address.
- The Aardvark adapter was unable to seize the bus due to the presence of another I²C master. **Here, the arbitration was lost during the slave addressing phase — results can be unpredictable**.
- The slave was addressed and no bytes were written to it because `num_bytes` was set to 0.

The number of bytes written can be less than the requested number of bytes in the transaction due to the following possibilities.

- The Aardvark adapter loses the bus during the data transmission due to the presence of another I²C master.
- The slave refuses the reception of any more bytes.

Master Write Extended (aa_i2c_write_ext)

```
int aa_i2c_write_ext (Aardvark          aardvark,
                      aa_u16            slave_addr,
                      AardvarkI2cFlags  flags,
                      aa_u16            num_bytes,
                      const aa_u08 *    data_out,
                      aa_u16 *          num_written);
```

Write a stream of bytes to the I²C slave device with extended status information.

Arguments

`aardvark`: handle of an Aardvark adapter

`slave_addr`: the slave from which to read

`flags`: special operations as described in "Notes" section

`num_bytes`: the number of bytes to write (maximum 65535)

`data_out`: pointer to data

`num_written`: the actual number of bytes written

Return Value

Status code (see "Notes" section).

Specific Error Codes

None.

Details

This function operates exactly like `aa_i2c_write`, except that the return value now specifies a status code. The number of bytes written is returned through an additional pointer argument at the tail of the parameter list.

The status code allows the user to discover specific events on the I²C bus that would otherwise be transparent given only the number of bytes transacted. The "Notes" section describes the status codes.

For a master write operation, the `AA_I2C_STATUS_DATA_NACK` flag can be useful in the following situation:

- Normally the I²C master will write to the slave until the slave issues a NACK or the requested number of bytes have been written.
- If the master has wishes to write 10 bytes, the I²C slave issues either an ACK or NACK on the tenth byte without affecting the total number of bytes transferred. Hence, the `aa_i2c_write` function cannot provide the caller with the information that the 10th byte was ACK'ed or NACK'ed.
- On the other hand, if the `aa_i2c_write_ext` is used, the status code will distinguish the two scenarios. This status information could be useful for further communications with that particular slave device.

Master Write-Read (aa_i2c_write_read)

```
int aa_i2c_write_read (Aardvark          aardvark,
                       aa_u16            slave_addr,
                       AardvarkI2cFlags  flags,
                       aa_u16            out_num_bytes,
                       const aa_u08 *    out_data,
                       aa_u16 *          num_written,
                       aa_u16            in_num_bytes,
                       aa_u08 *          in_data,
                       aa_u16 *          num_read);
```

Write a stream of bytes to the I²C slave device followed by a read from the same slave device.

Arguments

`aardvark`: handle of an Aardvark adapter

`slave_addr`: the slave from which to read

`flags`: special operations as described in "Notes" section

`out_num_bytes`: the number of bytes to write (maximum 65535)

`out_data`: pointer to data to write

`num_written`: the actual number of bytes written

`in_num_bytes`: the number of bytes to read (maximum 65535)

`in_data`: pointer to data for read

`num_read`: the actual number of bytes read

Return Value

Combined I²C status code from the write and read operations.

Specific Error Codes

None.

Details

This function performs the functions of `aa_i2c_write_ext` and `aa_i2c_read_ext` in one atomic operation, thereby minimizing the latency between consecutive write and read operations due to the USB communication layer.

A combined status code from the write and the read operations is provided as the return value of the function. The return value is constructed as `(read_status « 8) | (write_status)`. See the "Notes" section, along with the details of `aa_i2c_write_ext` and `aa_i2c_read_ext`, for more discussion about the specific status codes. Note that if the write phase of the operation completes with a non-zero status code, the Aardvark adapter will not physically execute the read phase of the operation.

If either the write or read fails with an error (as opposed to simply a non-zero status code), the return value of the function reflects the appropriate error code, with preference given to write errors.

I²C Slave

Slave Enable (aa_i2c_slave_enable)

```
int aa_i2c_slave_enable  (Aardvark  aardvark,
                          aa_u08    addr,
                          aa_u16    maxTxBytes,
                          aa_u16    maxRxBytes);
```

Enable the Aardvark adapter as an I²C slave device.

Arguments

`aardvark`: handle of an Aardvark adapter

`addr`: address of this slave

`maxTxBytes`: max number of bytes to transmit per transaction

`maxRxBytes`: max number of bytes to receive per transaction

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

The lower 7 bits of addr should correspond to the slave address of this Aardvark adapter. If the topmost bit of addr is set, the slave will respond to a general call transmission by an I²C master. After having been addressed by a general call, the Aardvark I²C slave treats the transaction no differently than a single slave communication. There is no support for the hardware address programming feature of the general call that is described in the I²C protocol specification since that capability is not needed for Aardvark devices.

If maxTxBytes is 0, there is no limit on the number of bytes that this slave will transmit per transaction. If it is non-zero, then the slave will stop transmitting bytes at the specified limit and subsequent bytes received by the master will be 0xff due to the bus pull-up resistors. The response that is transmitted by the slave is set through the aa_i2c_slave_set_response function described below. If the maximum is greater than the response (as set through aa_i2c_slave_set_response) the Aardvark slave will wrap the response string as many times as necessary to send the requested number of bytes.

If maxRxBytes is 0, the slave can receive an unlimited number of bytes from the master. However, if it is non-zero, the slave will send a not-acknowledge bit after the last byte that it accepts. The master should then release the bus. Even if the master does not stop transmitting, the slave will return the received data back to the host PC and then transition to a idle state, waiting to be addressed in a subsequent transaction.

It is never possible to *restrict* a transmit or receive to 0 bytes. Furthermore, once the slave is addressed by a master read operation it is always guaranteed to transmit at least 1 byte.

If a master transaction is executed after the slave features have been enabled, the slave features will remain enabled after the master transaction completes.

Slave Disable (aa_i2c_slave_disable)

```
int aa_i2c_slave_disable (Aardvark aardvark);
```

Disable the Aardvark adapter as an I²C slave device.

Arguments

aardvark: handle of an Aardvark adapter

Return Value

An Aardvark status code is returned that is AA_OK on success.

Specific Error Codes

None.

Details

None.

Slave Set Response (aa_i2c_slave_set_response)

```
int aa_i2c_slave_set_response (Aardvark        aardvark,
                               aa_u08          num_bytes,
                               const aa_u08 *  data_out);
```

Set the slave response in the event the Aardvark adapter is put into slave mode and contacted by a master.

Arguments

`aardvark`: handle of an Aardvark adapter

`num_bytes`: number of bytes for the slave response

`data_out`: pointer to the slave response

Return Value

The number of bytes accepted by the Aardvark slave for the response.

Specific Error Codes

None.

Details

The value of `num_bytes` must be greater than zero. If it is zero, the response string is undefined until this function is called with the correct parameters.

Due to limited buffer space on the Aardvark slave, the device may only accept a portion of the intended response. If the value returned by this function is less than `num_bytes` the Aardvark slave has dropped the remainder of the bytes.

If more bytes are requested in a transaction, the response string will be wrapped as many times as necessary to complete the transaction.

The buffer space will nominally be 64 bytes but can change depending on firmware revision.

Slave Write Statistics (aa_i2c_slave_write_stats)

```
int aa_i2c_slave_write_stats (Aardvark aardvark);
```

Return number of bytes written from a previous Aardvark I²C slave to I²C master transmission.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

The number of bytes written asynchronously.

Specific Error Codes

`AA_I2C_SLAVE_TIMEOUT`: There was no recent slave transmission.

Details

The transmission of bytes from the Aardvark slave, when it is configured as an I²C slave, is asynchronous with respect to the PC host software. Hence, there could be multiple responses queued up from previous write transactions.

This function will wait 500 milliseconds before timing out. See the `aa_async_poll` function if a variable timeout is required.

Slave Write Statistics Extended (aa_i2c_slave_write_stats_ext)

```
int aa_i2c_slave_write_stats_ext (Aardvark  aardvark,
                                  aa_u16 *  num_written);
```

Return number of bytes written from a previous Aardvark I²C slave to I²C master transmission with extended status information.

Arguments

aardvark: handle of an Aardvark adapter

num_written: : the number of bytes written by the slave

Return Value

Status code (see "Notes" section).

Specific Error Codes

None.

Details

This function operates exactly like aa_i2c_slave_write_stats, except that the return value now specifies a status code. The number of bytes written is returned through an additional pointer argument at the tail of the parameter list.

The only possible status code is AA_I2C_STATUS_BUS_ERROR which can occur when an illegal START, STOP, or RESTART condition appears on the bus during a transaction. In this case the num_written may not exactly reflect the number of bytes written by the slave. It can be off by 1.

Slave Read (aa_i2c_slave_read)

```
int aa_i2c_slave_read (Aardvark  aardvark,
                       aa_u08 *  addr,
                       aa_u16    num_bytes,
                       aa_u08 *  data_out);
```

Read the bytes from an I²C slave reception.

Arguments

aardvark: handle of an Aardvark adapter

addr: the address to which the received message was sent

num_bytes: the maximum size of the data buffer

data_out: pointer to data buffer

Return Value

This function returns the number of bytes read asynchronously.

Specific Error Codes

AA_I2C_SLAVE_TIMEOUT: There was no recent slave transmission.

AA_I2C_DROPPED_EXCESS_BYTES: The msg was larger than num_bytes.

Details

If the message was directed to this specific slave, *addr will be set to the value of this slave's address. However, this slave may have received this message through a general call addressing. In this case, *addr will be 0x80 instead of its own address.

The num_bytes parameter specifies the size of the memory pointed to by data. It is possible, however, that the received slave message exceeds this length. In such a situation, AA_I2C_DROPPED_EXCESS_BYTES is returned, meaning that num_bytes was placed into data but the remaining bytes were discarded.

There is no cause for alarm if the number of bytes read is less than num_bytes. This simply indicates that the incoming message was short.

The reception of bytes by the Aardvark slave, when it is configured as an I²C slave, is asynchronous with respect to the PC host software. Hence, there could be multiple responses queued up from previous transactions.

This function will wait 500 milliseconds before timing out. See the aa_async_poll function if a variable timeout is required.

Slave Read Extended (aa_i2c_slave_read_ext)

```
int aa_i2c_slave_read_ext (Aardvark  aardvark,
                           aa_u08 *  addr,
                           aa_u16    num_bytes,
                           aa_u08 *  data_out
                           aa_u16 *  num_read);
```

Read the bytes from an I²C slave reception with extended status information.

Arguments

aardvark: handle of an Aardvark adapter

addr: the address to which the received message was sent

num_bytes: the maximum size of the data buffer

data_out: pointer to data buffer

num_read: the actual number of bytes read by the slave

Return Value

Status code (see "Notes" section).

Specific Error Codes

None.

Details

This function operates exactly like aa_i2c_slave_read, except that the return value now specifies a status code. The number of bytes read is returned through an additional pointer argument at the tail of the parameter list.

The only possible status code is AA_I2C_STATUS_BUS_ERROR which can occur when an illegal START, STOP, or RESTART condition appears on the bus during a transaction.

5.6 SPI Interface

SPI Notes

1. The SPI master and slave must both be configured to use the same bit protocol (mode).
2. It is not necessary to set the bitrate for the Aardvark SPI slave.
3. An SPI master operation read or write operation can be transacted while leaving the SPI slave functionality enabled. During the master transaction, the slave will be temporarily deactivated. Once the master transaction is complete, the slave will be automatically reactivated.
4. It is always advisable to set the slave response before first enabling the slave. This ensures that valid data is sent to any requesting master.
5. It is not possible to receive messages larger than approximately 4 KiB as a slave due to operating system limitations on the asynchronous incoming buffer. As such, one should not queue up more than 4 KiB of total slave data between calls to the Aardvark API.
6. It is not possible to send messages larger than approximately 4 KiB as a master due to operating system limitations on the asynchronous incoming buffer. The SPI is full-duplex so there must be enough buffer space to accommodate the slave response when sending as a master.
7. Sending zero bytes as an SPI master will simply toggle the slave select line for 5–10 µs.

General SPI

Configure (aa_spi_configure)

```
int aa_spi_configure (Aardvark             aardvark,
                      AardvarkSpiPolarity  polarity,
                      AardvarkSpiPhase     phase,
                      AardvarkSpiBitorder  bitorder);
```

Configure the SPI master or slave interface.

Arguments

`aardvark`: handle of an Aardvark adapter

`polarity`: `AA_SPI_POL_RISING_FALLING` or `AA_SPI_POL_FALLING_RISING`

`phase`: `AA_SPI_PHASE_SAMPLE_SETUP` or `AA_SPI_PHASE_SETUP_SAMPLE`

`bitorder`: `AA_SPI_BITORDER_MSB` or `AA_SPI_BITORDER_LSB`

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

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

The polarity option specifies which transition constitutes the leading edge and which transition is the falling edge. For example, AA_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. For example, AA_SPI_PHASE_SAMPLE_SETUP would configure the SPI to sample on the leading edge and setup on the trailing edge.

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

The pair (AA_SPI_POL_FALLING_RISING, AA_SPI_PHASE_SETUP_SAMPLE) would correspond to mode 3 in the figure found in the "SPI Background" chapter.

SPI Master

Set Bitrate (aa_spi_bitrate)

```
int aa_spi_bitrate (Aardvark aardvark, int bitrate_khz);
```

Set the SPI bitrate in kilohertz.

Arguments

aardvark: handle of an Aardvark 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 1000 kHz.

Only certain discrete bitrates are supported by the Aardvark adapter. As such, this actual bitrate set will be less than or equal to the requested bitrate unless the requested value is less than 125 kHz, in which case the Aardvark adapter will default to 125 kHz.

If bitrate_khz is 0, the function will return the bitrate presently set on the Aardvark adapter and the bitrate will be left unmodified.

Set Slave Select Polarity (aa_spi_master_ss_polarity)

```
int aa_spi_master_ss_polarity (Aardvark                aardvark,
                               AardvarkSpiSSPolarity   polarity);
```

Change the output polarity on the SS line.

Arguments

aardvark: handle of an Aardvark adapter

polarity: AA_SPI_SS_ACTIVE_LOW or AA_SPI_SS_ACTIVE_HIGH

Return Value

An Aardvark status code is returned that is AA_OK on success.

Specific Error Codes

None.

Details

This function only affects the SPI master functions on the Aardvark adapter. When configured as an SPI slave, the Aardvark adapter will always be setup with SS as active low.

Master Write/Read (aa_spi_write)

```
int aa_spi_write (Aardvark        aardvark,
                  aa_u16          out_num_bytes,
                  const aa_u08 *  data_out,
                  aa_u16          in_num_bytes,
                  aa_u08 *        data_in);
```

Write a stream of bytes to the downstream SPI slave device and read back the full-duplex response.

Arguments

aardvark: handle of an Aardvark adapter

out_num_bytes: number of bytes to send

data_out: pointer to the bytes to transmit out

in_num_bytes: number of bytes to receive

data_in: pointer to memory that will hold the received data

Return Value

This function returns the total number of bytes read from the slave which normally will be the same as the number of bytes written to the slave. See below for how this value relates to in_num_bytes.

Specific Error Codes

AA_SPI_WRITE_ERROR: There was an error writing to the Aardvark adapter. This is most likely a result of a communication error. Make sure that out_num_bytes is less than 4 KiB.

Details

Due to the full-duplex nature of the SPI protocol, for every byte written to the slave, one byte is also received. The Aardvark will always receive the same number of bytes that it sends out (barring any error). This is the return value mentioned above. The user has the option of saving all, some, or none of those received bytes by varying the size of in_num_bytes.

This function will always write out the number of bytes defined by out_num_bytes from the memory pointed to by data_out. When out_num_bytes is larger than in_num_bytes, data_in is completely filled and any extra bytes are dropped. When out_num_bytes is less than in_num_bytes, all the received bytes are saved and data_in is only partially filled.

The data_in pointer should reference memory that is at least allocated to the size specified by in_num_bytes. Otherwise there will be a memory access violation in the program.

If out_num_bytes is 0, no bytes will be written to the slave. However, the slave select line will be dropped for 5–10 µs. This can be useful in sending a signal to a downstream SPI slave

without actually sending any bytes. For example, if an SPI slave has tied the slave select to an interrupt line and it sees the line is toggled without any bytes sent, it can interpret the action as a command to prepare its firmware for an subsequent reception of bytes. If `out_num_bytes` is 0, `data_out`, `data_in`, and `in_num_bytes` can be set to 0.

If the return value of this function is less than `out_num_bytes`, there was an error. SPI is a bit-blasting scheme where the master does not even know if there is a slave on the other end of the transmission. Therefore, it is always expected that the master will send the entire length of the transaction.

An error will likely occur if the number of bytes sent is significantly greater than 4 KiB. This function cannot reliably execute larger transfers due to the buffering issues explained in the "Software | Application Notes" section. Only a partial number of bytes will be sent to the slave and only a partial number will be received from the slave; it is quite possible that these numbers will not be equal. The size of the partial response is returned by this function and any received data up to `in_num_bytes` will be in the memory pointed to by `data_in`. Note that the last few bytes of the response may be corrupted as well.

SPI Slave

Slave Enable (aa_spi_slave_enable)

```
int aa_spi_slave_enable (Aardvark aardvark);
```

Enable the Aardvark as an SPI slave device.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

None.

Slave Disable (aa_spi_slave_disable)

```
int aa_spi_slave_disable (Aardvark aardvark);
```

Disable the Aardvark as an SPI slave device.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

None.

Slave Set Response (aa_spi_slave_set_response)

```
int aa_spi_slave_set_response (Aardvark        aardvark,
                               aa_u08          num_bytes,
                               const aa_u08 *  data_out);
```

Set the slave response in the event the Aardvark adapter is put into slave mode and contacted by a master.

Arguments

`aardvark`: handle of an Aardvark adapter

`num_bytes`: number of bytes for the slave response

`data_out`: pointer to the slave response

Return Value

The number of bytes accepted by the Aardvark for the response.

Specific Error Codes

None.

Details

The value of `num_bytes` must be greater than zero. If it is zero, the response string is undefined until this function is called with the correct parameters.

Due to limited buffer space on the Aardvark slave, the device may only accept a portion of the intended response. If the value returned by this function is less than `num_bytes` the Aardvark slave has dropped the remainder of the bytes.

If more bytes are requested in a transaction, the response string will be wrapped as many times as necessary to complete the transaction.

The buffer space will nominally be 64 bytes but may change depending on firmware revision.

Slave Read (aa_spi_slave_read)

```
int aa_spi_slave_read (Aardvark   aardvark,
                       aa_u16     num_bytes,
                       aa_u08 *   data_in);
```

Read the bytes from an SPI slave reception.

Arguments

`aardvark`: handle of an Aardvark adapter

`num_bytes`: the maximum size of the data buffer

`data_in`: pointer to data buffer

Return Value

This function returns the number of bytes read asynchronously.

Specific Error Codes

`AA_SPI_SLAVE_TIMEOUT`: There was no recent slave transmission.

`AA_SPI_DROPPED_EXCESS_BYTES`: The msg was larger than num_bytes.

Details

The num_bytes parameter specifies the size of the memory pointed to by data. It is possible, however, that the received slave message exceeds this length. In such a situation, AA_SPI_DROPPED_EXCESS_BYTES is returned, meaning that num_bytes was placed into data but the remaining bytes were discarded.

There is no cause for alarm if the number of bytes read is less than num_bytes. This simply indicates that the incoming message was short.

The reception of bytes by the Aardvark adapter, when it is configured as an SPI slave, is asynchronous with respect to the PC host software. Hence, there could be multiple responses queued up from previous write transactions.

This function will wait 500 milliseconds before timing out. See the aa_async_poll function if a variable timeout is required.

The SPI API does not include a function that is analogous to the I²C function aa_i2c_slave_write_stats. Since SPI is a full-duplex standard, the slave writes to the master whenever it receives bytes from the master. Hence, a received message from aa_i2c_slave_read implies that an equal number of bytes were sent to the master.

5.7 GPIO Interface

GPIO Notes

1. The following enumerated type maps the named lines on the Aardvark I²C /SPI output cable to bit positions in the GPIO API. All GPIO API functions will index these lines through a single 8-bit masked value. Thus, each bit position in the mask can be referred back its corresponding line through the mapping described below.

Table 8: `AardvarkGpioBits`*: enumerated type of line locations in bit mask*

AA_GPIO_SCL	Pin 1	0x01	I²C SCL line
AA_GPIO_SDA	Pin 3	0x02	I²C SDA line
AA_GPIO_MISO	Pin 5	0x04	SPI MISO line
AA_GPIO_SCK	Pin 7	0x08	SPI SCK line
AA_GPIO_MOSI	Pin 8	0x10	SPI MOSI line
AA_GPIO_SS	Pin 9	0x20	SPI SS line

2. There is no check in the GPIO API calls to see if a particular GPIO line is enabled in the current configuration. If a line is not enabled for GPIO, the get function will simply return 0 for those bits. Another example is if one changes the GPIO directions for I²C lines while the I²C subsystem is still active. These new direction values will be cached and will automatically be activate if a later call to aa_configure disables the I²C subsystem and enables GPIO for the I²C lines. The same type of behavior holds for `aa_gpio_pullup` and `aa_gpio_set`.

3. Additionally, for lines that are not configured as inputs, a change in the pullup configuration using `aa_gpio_pullup` will be cached and will take effect the next time the line is active and configured as an input. The same behavior holds for `aa_gpio_set` when a line is configured as an input instead of an output.

4. On Aardvark adapter power-up, directions default to all input, pullups default to disabled and outputs default to logic low. Also the GPIO subsystem is off by default. It must be activated by using `aa_configure`.

5. Note: For hardware version 1.02, it is not possible to have high-Z inputs on the I²C lines since that hardware has 2.2K pullups on the I²C bus.

GPIO Interface

Direction (aa_gpio_direction)

```
int aa_gpio_direction (Aardvark aardvark,
                       aa_u08   direction_mask);
```

Change the direction of the GPIO lines between input and output directions.

Arguments

`aardvark`: handle of an Aardvark adapter

`direction_mask`: each bit corresponds to the physical line as given by `AardvarkGpioBits`. If a line's bit is 0, the line is configured as an input. Otherwise it will be an output.

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

None.

Pullup (aa_gpio_pullup)

```
int aa_gpio_pullup (Aardvark aardvark,
                    aa_u08   pullup_mask);
```

Change the pullup configuration of the GPIO lines.

Arguments

`aardvark`: handle of an Aardvark adapter

`pullup_mask`: each bit corresponds to the physical line as given by `AardvarkGpioBits`. If a line's bit is 1, the line's pullup is active whenever the line is configured as an input. Otherwise the pullup will be deactivated.

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

None.

Get (aa_gpio_get)

```
int aa_gpio_get (Aardvark aardvark);
```

Get the value of current GPIO inputs.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

An integer value, organized as a bitmask in the fashion described by `AardvarkGpioBits`. Any line that is logic high will have a its corresponding bit active. If the line is logic low the bit will not be active in the bit mask.

Specific Error Codes

None.

Details

A line's bit position in the mask will be 0 if it is configured as an output or if it corresponds to a subsystem that is still active.

Set (aa_gpio_set)

```
int aa_gpio_set (Aardvark aardvark,
                 aa_u08   value);
```

Set the value of current GPIO outputs.

Arguments

`aardvark`: handle of an Aardvark adapter

`value`: a bitmask specifying which outputs should be set to logic high and which should be set to logic low.

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

If a line is configured as an input or not activated for GPIO, the output value will be cached. The next time the line is an output and activated for GPIO, the output value previously set will automatically take effect.

Change (aa_gpio_change)

```
int aa_gpio_change (Aardvark aardvark,
                    aa_u16   timeout);
```

Block until there is a change on the GPIO input lines.

Arguments

`aardvark`: handle of an Aardvark adapter

`timeout`: time to wait for a change in milliseconds

Return Value

The current state of the GPIO input lines.

Specific Error Codes

None.

Details

The function will return either when a change has occurred or the timeout expires. Pins configured for I²C or SPI will be ignored. Pins configured as outputs will be ignored. The timeout, specified in milliseconds, has a precision of approximately 2 ms. The maximum allowable timeout is approximately 60 seconds. If the timeout expires, this function will return the current state of the GPIO lines. It is the application's responsibility to save the old value of the lines and determine if there is a change based on the return value of this function.

The function `aa_gpio_change` will return immediately with the current value of the GPIO lines for the first invocation after any of the following functions are called: `aa_configure`, `aa_gpio_direction`, or `aa_gpio_pullup`. If the function `aa_gpio_get` is called before calling `aa_gpio_change`, `aa_gpio_change` will only register any changes from the value last returned by `aa_gpio_get`.

5.8 I²C Bus Monitor

Monitor Notes

1. The Aardvark adapter can continuously monitor I²C traffic up to 125 khz. It may be possible to monitor higher bitrates for small transactions, but the reliability is not guaranteed.
2. Activating the bus monitor will disable all other functions on the Aardvark adapter and flush all pending asynchronous messages. The Aardvark adapter will be placed into the `AA_CONFIG_GPIO_ONLY` mode by default. Once the bus monitor is disabled, the user must use `aa_configure` to re-enable either I²C or SPI functions if these operations are required.
3. Once the bus monitor is enabled, the execution of any other communication function on the Aardvark adapter will disable the bus monitor.
4. The I²C pullup resistors can be enabled or disabled during I²C monitor use. Simply use the `aa_i2c_pullup` function before enabling the bus monitor.
5. Once enabled, the I²C monitor automatically monitors the attached I²C bus and returns data back to the host asynchronously. Due to operating system limitations on the asynchronous incoming buffer, one must ensure that the asynchronous data is serviced regularly. As such, one should not queue up more than 4 KiB of total monitor data between calls to the Aardvark API. For example, problems can arise when there is a high flow of traffic on the bus that is being monitored.

I²C Bus Monitor Interface

Monitor Enable (aa_i2c_monitor_enable)

```
int aa_i2c_monitor_enable (Aardvark aardvark);
```

Enable the Aardvark adapter as an I²C monitoring device.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

This function has a 100 ms delay to flush all communications buffers.

Monitor Disable (aa_i2c_monitor_disable)

```
int aa_i2c_monitor_disable (Aardvark aardvark);
```

Disable the Aardvark adapter as an I²C monitoring device.

Arguments

`aardvark`: handle of an Aardvark adapter

Return Value

An Aardvark status code is returned that is `AA_OK` on success.

Specific Error Codes

None.

Details

This function has a 100 ms delay to flush all communications buffers.

Monitor Read (aa_i2c_monitor_read)

```
int aa_i2c_monitor_read (Aardvark  aardvark,
                         aa_u16    num_bytes,
                         aa_u16 *  data);
```

Read the bytes from an I²C monitor operation.

Arguments

`aardvark`: handle of an Aardvark adapter

`num_bytes`: the maximum size of the data buffer

`data`: pointer to data buffer

Return Value

This function returns the number of monitor bytes read.

Specific Error Codes

`AA_I2C_MONITOR_NOT_ENABLED`: The monitor functionality was not enabled.

Details

Once enabled, the I²C monitor automatically monitors the attached I²C bus and returns data back to the host asynchronously. This function allows the PC to retrieve the asynchronous data into a buffer for analysis.

The monitored data is returned in a semi-raw format. The returned data buffer contains all information about bit-level signaling on the bus. Parsing this information is simple and the user is referred to the examples that are included with the Aardvark software distribution. Furthermore, the Control Center application demonstrates a GUI that uses this monitoring function.

The information in each element of the returned buffer is marked as follows:

1. Element is equal to the constant `AA_I2C_MONITOR_CMD_START` (`0xff00`) if a start bit has been encountered.
2. Element is equal to the constant `AA_I2C_MONITOR_CMD_STOP` (`0xff01`) if a stop bit has been encountered.
3. If the element is neither a start or stop bit, the bottom 8-bits are equal to the 8 data bits of the I²C transaction.
4. The element can have the 9th bit equal to 1 if the data byte was not acknowledged by the I²C receiving device. One can test for this condition by bitwise AND'ing the element with `AA_I2C_MONITOR_NACK` (`0x100`).

The monitor function returns one I²C transaction at a time. It expects that every transaction starts with an I²C START bus signal. The function returns when encountering a STOP command. The application can then process the transaction and call the monitor function again to process a subsequent transaction.

In the event of a repeated start on the bus, the incoming buffer will contain a START code without a prior STOP code. All traffic between the initial START command and the final STOP command will be returned in one call to this function.

As per the I²C protocol, the slave address is the first byte transmitted after the START command. The user can analyze this byte to determine the slave address for the bus transaction as well as the direction of the transaction (i.e., master read versus master write). The top 7 bits of the byte correspond to the slave address and the least significant bit denotes the transfer direction. See the Philips I²C protocol definition for more details.

There can be bit alignment issues if the monitor is plugged into a bus that is already in the middle of an I²C transaction. The data will most likely be corrupted. Likewise, there can be corruption if there is an overflow of the operating system's receive buffer.

This function will wait 500 milliseconds before timing out. See the aa_async_poll function if a variable timeout is required.

5.9 Error Codes

Table 9: *Aardvark API Error Codes*

Literal Name	Value	aa_status_string() return value
AA_OK	0	ok
AA_UNABLE_TO_LOAD_LIBRARY	-1	unable to load library
AA_UNABLE_TO_LOAD_DRIVER	-2	unable to load USB driver
AA_UNABLE_TO_LOAD_FUNCTION	-3	unable to load binding function
AA_INCOMPATIBLE_LIBRARY	-4	incompatible library version
AA_INCOMPATIBLE_DEVICE	-5	incompatible device version
AA_COMMUNICATION_ERROR	-6	communication error
AA_UNABLE_TO_OPEN	-7	unable to open device
AA_UNABLE_TO_CLOSE	-8	unable to close device
AA_INVALID_HANDLE	-9	invalid device handle
AA_CONFIG_ERROR	-10	configuration error
AA_I2C_NOT_AVAILABLE	-100	i2c feature not available
AA_I2C_NOT_ENABLED	-101	i2c not enabled
AA_I2C_READ_ERROR	-102	i2c read error
AA_I2C_WRITE_ERROR	-103	i2c write error
AA_I2C_SLAVE_BAD_CONFIG	-104	i2c slave enable bad config
AA_I2C_SLAVE_READ_ERROR	-105	i2c slave read error
AA_I2C_SLAVE_TIMEOUT	-106	i2c slave timeout
AA_I2C_DROPPED_EXCESS_BYTES	-107	i2c slave dropped excess bytes
AA_I2C_BUS_ALREADY_FREE	-108	i2c bus already free
AA_SPI_NOT_AVAILBLE	-200	spi feature not available
AA_SPI_NOT_ENABLED	-201	spi not enabled
AA_SPI_WRITE_ERROR	-202	spi write error
AA_SPI_SLAVE_READ_ERROR	-203	spi slave read error
AA_SPI_SLAVE_TIMEOUT	-204	spi slave timeout
AA_SPI_DROPPED_EXCESS_BYTES	-205	spi slave dropped excess bytes
AA_GPIO_NOT_AVAILABLE	-400	gpio feature not available
AA_I2C_MONITOR_NOT_AVAILABLE	-500	i2c bus monitor feature not available
AA_I2C_MONITOR_NOT_ENABLED	-501	i2c bus monitor 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