

cyan technology



**eCOG1X USB Library**  
**Version 1.0**

**Cyan Technology**

**eCOG1X**  
**USB Library**  
**V1.0**

**1 March 2007**



## Confidential and Proprietary Information

© Cyan Technology Ltd., 2006-2007

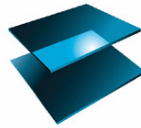
This document contains confidential and proprietary information of Cyan Technology Ltd. and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd. recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd. in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd. shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd. specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.



c y a n   t e c h n o l o g y

## Revision History

| Version | Date       | Notes          |
|---------|------------|----------------|
| V1.0    | 01/03/2007 | First release. |
|         |            |                |
|         |            |                |
|         |            |                |

# Contents

|  |      |
|--|------|
| List of Figures . . . . .  | v    |
| List of Tables . . . . .   | vii  |
| 1 Introduction . . . . .   | 1-1  |
| 1.1 Additional Documents . . . . .                               | 1-1  |
| 1.2 Typographical Conventions . . . . .                          | 1-1  |
| 1.3 Part Identification . . . . .                                | 1-1  |
| 1.4 Disclaimer . . . . .   | 1-1  |
| 1.5 Glossary . . . . .   | 1-2  |
| 1.6 Registers and Bit Fields . . . . .                           | 1-2  |
| 2 Overview . . . . .   | 2-1  |
| 2.1 Background . . . . .   | 2-1  |
| 2.2 Use With CyanIDE . . . . .                                   | 2-1  |
| 2.3 Hardware Configuration . . . . .                             | 2-2  |
| 2.4 Internal USB PHY . . . . .                                   | 2-3  |
| 2.5 External ULPI PHY . . . . .                                  | 2-3  |
| 2.6 Endpoints . . . . .  | 2-4  |
| 2.7 USB Core Register Access . . . . .                           | 2-4  |
| 2.8 Endian Issues . . . . .                                      | 2-4  |
| 2.9 Byte Accesses . . . . .                                      | 2-4  |
| 2.10 Byte Packing . . . . .                                      | 2-5  |
| 2.11 Endpoint Directions . . . . .                               | 2-5  |
| 2.12 Memory Configuration . . . . .                              | 2-6  |
| 2.13 Background Timer . . . . .                                  | 2-6  |
| 3 Writing Software . . . . .                                     | 3-1  |
| 3.1 The Library Model . . . . .                                  | 3-1  |
| 3.2 Including The USB Library In A Project . . . . .             | 3-2  |
| 3.3 Library Plugins . . . . .                                    | 3-2  |
| 3.4 Adding USB Functionality To Existing Code . . . . .          | 3-2  |
| 3.5 DMA and FIFO Transfers . . . . .                             | 3-3  |
| 3.6 Configuring the eCOG1X for USB . . . . .                     | 3-3  |
| 3.7 Host, Peripheral and OTG Operation . . . . .                 | 3-4  |
| 3.8 Interrupts and Events . . . . .                              | 3-6  |
| 3.9 Configuring Endpoints . . . . .                              | 3-9  |
| 3.10 Transferring Data Using The Default Endpoint, EP0 . . . . . | 3-10 |
| 3.11 Transferring Data Using Endpoints EP1-EP3 . . . . .         | 3-11 |
| 3.12 Suspending the USB Core . . . . .                           | 3-11 |

|      |   |      |
|------|---|------|
| 4    | Plugins . . . . .                               | 4-1  |
| 4.1  | What Plugins Offer . . . . .                    | 4-1  |
| 4.2  | Using The Plugin Architecture . . . . .         | 4-2  |
| 4.3  | Keyboard Peripheral Plugin . . . . .            | 4-3  |
| 4.4  | Mass Storage Device Peripheral Plugin . . . . . | 4-4  |
| 4.5  | Audio Device Peripheral Plugin . . . . .        | 4-10 |
| 4.6  | Keyboard Host Plugin . . . . .                  | 4-12 |
| 4.7  | Mass Storage Device Host Plugin . . . . .       | 4-13 |
| 5    | Examples . . . . .                              | 5-1  |
| 5.1  | Peripheral Examples . . . . .                   | 5-1  |
| 5.2  | Host Examples . . . . .                         | 5-3  |
| 6    | Library Functions . . . . .                     | 6-1  |
| 6.1  | Library Utilities . . . . .                     | 6-1  |
| 6.2  | USB Setup . . . . .                             | 6-3  |
| 6.3  | Default Endpoint Data . . . . .                 | 6-4  |
| 6.4  | Endpoints 1-3 Data . . . . .                    | 6-6  |
| 6.5  | DMA Data Transfers . . . . .                    | 6-10 |
| 6.6  | Interrupts . . . . .                            | 6-11 |
| 6.7  | Events . . . . .                                | 6-13 |
| 6.8  | Standard Requests . . . . .                     | 6-15 |
| 6.9  | Plugins . . . . .                               | 6-17 |
| 6.10 | MSD Class Specific Functions . . . . .          | 6-18 |
| 6.11 | HID Class Specific Functions . . . . .          | 6-19 |
| 6.12 | Helper Routines . . . . .                       | 6-21 |

## List of Figures

|    |   |     |
|----|---|-----|
| 1: | USB core interfaces. . . . .                    | 2-2 |
| 2: | Connections for the internal USB PHY . . . . .  | 2-3 |
| 3: | Connections for the external ULPI PHY . . . . . | 2-3 |
| 4: | Endpoint buffer memory map . . . . .            | 2-6 |
| 5: | USB library architecture. . . . .               | 3-1 |
| 6: | A-device state machine . . . . .                | 3-4 |
| 7: | B-device state machine . . . . .                | 3-5 |





# List of Tables

|     |  |      |
|-----|--|------|
| 1:  | Glossary . . . . .                             | 1-2  |
| 2:  | Endpoint capabilities . . . . .                | 2-4  |
| 3:  | Comparison of FIFO and DMA transfers . . . . . | 3-3  |
| 4:  | Event types . . . . .                          | 3-8  |
| 5:  | File system structure . . . . .                | 4-8  |
| 6:  | Root directory structure . . . . .             | 4-9  |
| 7:  | Directory structure . . . . .                  | 4-9  |
| 8:  | Current directory structure . . . . .          | 4-9  |
| 9:  | Parent directory structure . . . . .           | 4-9  |
| 10: | USB core OTG states . . . . .                  | 6-1  |
| 11: | USB bus reset duration values . . . . .        | 6-3  |
| 12: | EP1-3 configuration parameters . . . . .       | 6-6  |
| 13: | EP1-3 enable parameters . . . . .              | 6-6  |
| 14: | DMA configuration parameters . . . . .         | 6-10 |
| 15: | USB core interrupt clear parameters . . . . .  | 6-11 |
| 16: | Check event parameters . . . . .               | 6-13 |
| 17: | Keycode modifiers . . . . .                    | 6-19 |



# 1 Introduction

This document is a reference manual for the eCOG1X USB library. The library provides comprehensive software support for the USB core, implemented in the Cyan Technology eCOG1X family of 16-bit microcontrollers.

## 1.1 Additional Documents

1. eCOG1X User Manual
2. eCOG1X USB Core User Manual
3. eCOG1 C-Compiler User Manual
4. eCOG1 Macro Assembler User Manual
5. CyanIDE User Manual

## 1.2 Typographical Conventions

|                                      |   |
|--------------------------------------|---|
| <b>bold</b>                          | Indicates an internal signal.   |
| <b><i>bold_italic</i></b>            | Indicates the name of a register.   |
| <b><i>bold_italic.dot</i></b>        | Indicates the name of a register field or a bit within a register.<br>There may be several items separated by dots with each item containing items to its right and being contained by items to its left. |
| <b>0x or H' prefix</b>               | Indicates a hexadecimal number.<br>Examples: 0xFF (= 255 decimal), H'0A (= 10 decimal).   |
| <b>Single quotes<br/>or b suffix</b> | Indicates a binary number.<br>Examples: 1000b (= 8 decimal), '11' (= 3 decimal).  |



*Notes within this document are used to highlight related or additional information to the user which is of interest or can prevent incorrect or undesirable operation. These notes are identified by the symbol shown on the left of this paragraph.*

## 1.3 Part Identification

In this document any reference to eCOG1X means the generic chip and is applicable to all versions. All eCOG1X devices are suffixed according to their version. Any reference to a particular device such as eCOG1X0A5 is specific to that version.

## 1.4 Disclaimer

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to life support equipment or systems intended to be surgically implanted into the body or any other life-critical application, whose failure to perform per documented instructions, can be reasonably expected to cause loss of life or significant injury. Cyan specifically disclaims any express or implied warranty of fitness for any or all of such uses.

## 1.5 Glossary

|       |   |
|-------|---|
| CPU   | Central Processing Unit                 |
| DMA   | Direct Memory Access                    |
| eCOG1 | Cyan Technology target micro controller |
| FIFO  | First-In First-Out buffer (queue)       |
| FSM   | Finite State Machine                    |
| HID   | Human Interface Device                  |
| HNP   | Host Negotiation Protocol               |
| IRAM  | Internal RAM                            |
| IROM  | Internal ROM (flash memory)             |
| IRQ   | Interrupt Request                       |
| ISR   | Interrupt Service Routine               |
| MMU   | Memory Management Unit                  |
| MSD   | Mass Storage Device                     |
| OTG   | On-The-Go (USB)                         |
| PHY   | Physical layer transceiver device       |
| RAM   | Random Access Memory                    |
| ROM   | Read Only Memory                        |
| SRP   |   |
| SSM   | System Support Module                   |
| ULPI  | USB Low Pin count Interface             |
| USB   | Universal Serial Bus                    |

Table 1: Glossary

## 1.6 Registers and Bit Fields

The on-chip I/O registers may be accessed as complete registers, or as named bit fields within the registers. The CyanIDE development tools provide the include file *registers.h* which contains both structure definitions for all on-chip registers and bit fields, and mask symbols for use within the registers. To access any register, use the structure prefix **rg**. To access a bit field within a register, use the structure prefix **fd**. For example:

```
// Write to flash program data register
rg.flash_prq_data = 0xA55A;

// Write to period bit field in configuration register
fd.flash_prq_cfg.period = 0x01F1;
```

This convention for accessing the peripheral registers is used in all the source code examples in this document.

Note that using the **fd** prefix to access bit fields within the registers generates a read-modify-write code sequence, which reads the complete register, modifies the selected bits, then writes the modified data back to the complete register. In some circumstances it may be preferable to avoid the read-modify-write sequence by writing explicitly a bit pattern value to the complete register.

## 2 Overview

The USB support library offers the following features:

- Configuration and setup is done within CyanIDE.
- USB host, peripheral and On-The-Go operation.
- Low speed, full speed and high speed transfers.
- Event driven messaging model, with event queue.
- Low level access to USB core hardware, if required.
- Integration with eCOG1X DMA peripheral for memory transfers.
- Plugins for common USB applications, requiring only minor custom configuration.
- Small code footprint (approx. 6Kwords for a keyboard, 10Kwords for an MSD).
- Small RAM footprint (approx. 1Kwords for a keyboard, 1.5Kwords for an MSD).
- Full source code provided, royalty free.
- Example CyanIDE applications projects provided.

### 2.1 Background

This document assumes that the reader is familiar with the USB Specification. No attempt is made in this document to explain basic USB operation or transfer protocol.

All the relevant USB specification documents are available from the USB Implementers Forum (USB-IF) at:

[www.usb.org](http://www.usb.org)

A good introductory document about the USB format is “USB In A Nutshell”, currently available from:

[www.beyondlogic.org](http://www.beyondlogic.org)

When developing USB mass storage devices (MSD), a subset of the SCSI command set is used. The SCSI documents are available from:

[www.t10.org](http://www.t10.org)

A good book on the subject is Jan Axelson’s “USB Complete”; more details at:

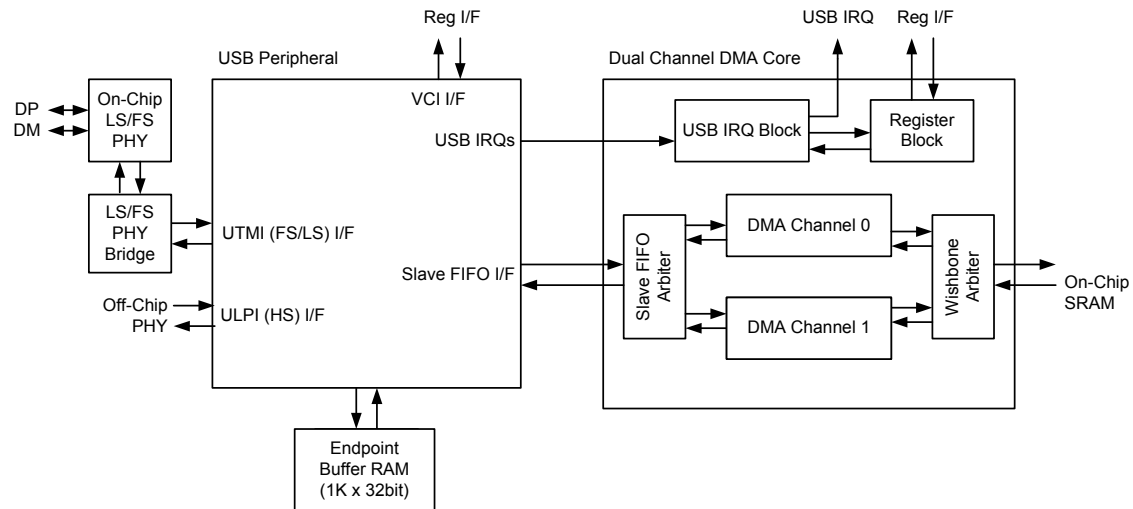
[www.lvr.com](http://www.lvr.com)

### 2.2 Use With CyanIDE

Much of the setup of the USB core can be performed with CyanIDE, which provides a user friendly interface for setting the important parameters for the USB core. Please consult the CyanIDE documentation for details on the configuration options available for the USB core.

## 2.3 Hardware Configuration

The USB core interfaces to both the eCOG1X's internal memory and DMA block.



**Figure 1: USB core interfaces**

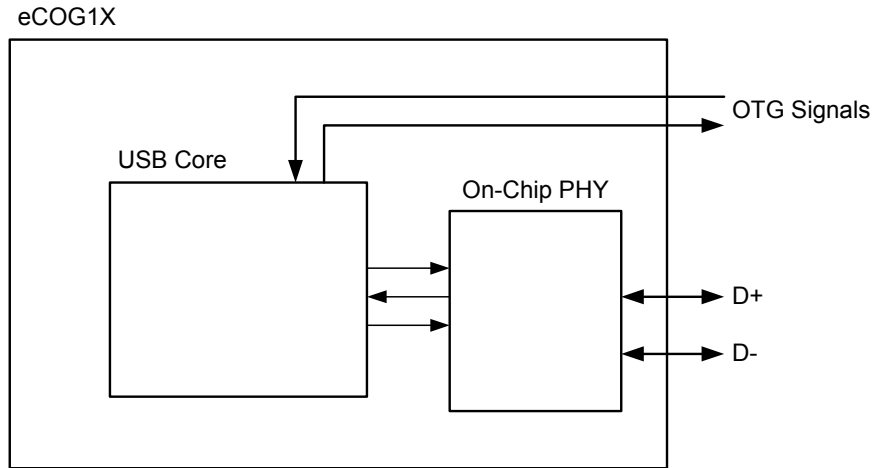
The USB core requires 4Kbytes of working memory, which is used for the endpoint buffers (see section 2.12). This is taken from the top of the internal memory and cannot then be accessed directly by the processor. Reading and writing to this memory is always done either through the USB core's FIFO registers, or with the DMA peripheral and the slave FIFO.

The USB core is controlled through a set of memory mapped registers, located at the address defined by `USB_BASE_ADDRESS` in the include file `usb_lib.h`. A detailed description of the USB core registers is given in the USB Core User Manual.

The DMA peripheral is controlled through the eCOG1X internal peripheral registers, accessed by the standard **rg** and **fd** register and bit field structures.

## 2.4 Internal USB PHY

The eCOG1X has an internal USB PHY which provides the necessary D+ and D- data signals. These signals include the necessary pull-up and pull-down resistors for operation and for speed detection. When operating as a low speed or full speed peripheral, this is all that is needed.



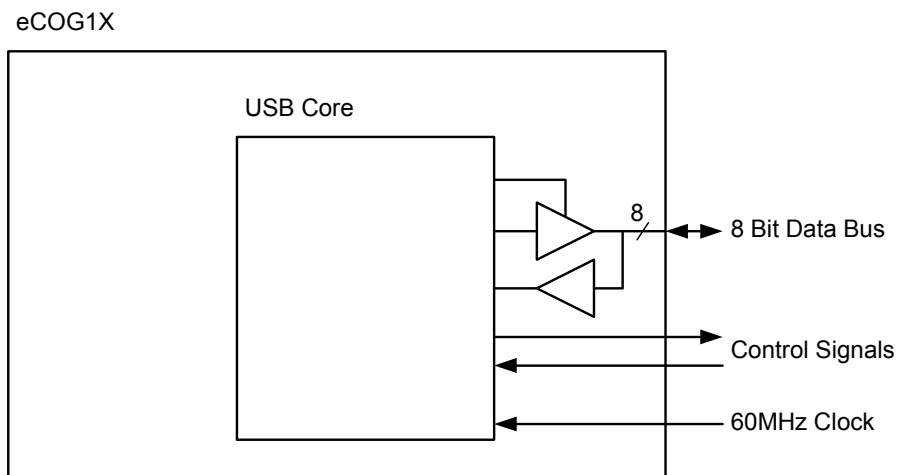
**Figure 2: Connections for the internal USB PHY**

To operate as a low speed or full speed host, a suitable Vbus power supply must be generated. This is done using an external device, such as the Maxim MAX3355E.

The internal PHY supports low speed and full speed transfers only.

## 2.5 External ULPI PHY

In order to operate at high speed, an external PHY is required that communicates with the eCOG1X over a ULPI interface.



**Figure 3: Connections for the external ULPI PHY**

## 2.6 Endpoints

The USB specification allows up to 16 endpoints, including the default EP0 control endpoint. The USB core in the eCOG1X supports EP0 + EP1-EP3, with differing capabilities. Each endpoint's capabilities are shown in the table below:

| Endpoint | Size (bytes) | Transfers        | Buffering                 |
|----------|--------------|------------------|---------------------------|
| EP0      | 64           | Control          | single buffered           |
| IN1      | 128          | BULK / INT / ISO | single or double buffered |
| OUT1     | 128          | BULK / INT / ISO | single or double buffered |
| IN2      | 256          | BULK / INT / ISO | single or double buffered |
| OUT2     | 256          | BULK / INT / ISO | single or double buffered |
| IN3      | 1024         | BULK / INT / ISO | single buffered           |
| OUT3     | 1024         | BULK / INT / ISO | single buffered           |

**Table 2: Endpoint capabilities**

Data is transferred using endpoints 1-3 using the provided FIFO registers or DMA peripheral. EP0 has 64 bytes of directly word-addressable data space in the USB core registers. See the USB Core User Manual for more details.

## 2.7 USB Core Register Access

There are two ways to access the USB core registers, either using `usb_reg_t` or `usb_fld_t` structure pointers. These can be set up as follows:

```
usb_reg_t* usb_rg = (usb_reg_t*)USB_BASE_ADDRESS;
usb_fld_t* usb_fd = (usb_fld_t*)USB_BASE_ADDRESS;
```

`usb_reg_t` provides access to the USB core registers as (unsigned) 16 bit values. `usb_fld_t` provides access as bit fields within the 16 bit registers. These two access methods are analogous to the **rg** and **fd** structures for the main eCOG1X registers.



**Note:** In most cases, `usb_fd` gives the most convenient way to access the USB core registers. Be aware though that bit field accesses on the eCOG1X are executed as read-modify-write operations on the 16 bit registers. In some cases, this can be important. For example, when clearing an interrupt register, it is not safe to do:

```
usb_fd->otg_int_sts.srpdet = 1;
```

to clear a single interrupt; other interrupts may be being reported as having fired in the same `otgstate_irq` register and the read-modify-write operation clears all of them with the above code. The correct code is:

```
usb_rg->otg_int_sts = USB_OTG_INT_STS_SRPDET_MASK;
```

which sets only a single bit and clears a single interrupt.

## 2.8 Endian Issues

All word values in the USB core registers are big-endian, the same as the eCOG1X.

## 2.9 Byte Accesses

The USB core registers do not support byte read accesses, so it is not possible to read the USB control registers a single byte at a time. Although byte write accesses are valid, they are not recommended unless absolutely necessary.

If you need to access individual bytes, you should do this using bit fields in a 16 bit word.



## 2.10 Byte Packing

The eCOG1x compiler always aligns word values on word boundaries, and this may not be what is required by the USB format. For example:

```
struct
{
    char a;           // Byte address 0x00
    unsigned int b;   // Byte address 0x02
};
```

Note how the compiler adds a packing byte between the two structure members. If it is required to create a structure which has word values aligned to odd byte addresses, then it is necessary to use bit fields to achieve this.

## 2.11 Endpoint Directions

The USB standard defines the direction of transfer as relative to the host. This is clear when the system is viewed as a whole, but it can be confusing when only one side of the transfer is considered.

When configuring endpoints in the USB core, directions are always given in terms of the direction of data flow. The following examples clarify this:

If a USB *host* wishes to configure an endpoint to *send* data, it configures the endpoint as *USB\_EPDIRN\_OUT*. It sends OUT packets.

If a USB *host* wishes to configure an endpoint to *receive* data, it configures the endpoint as *USB\_EPDIRN\_IN*. It sends IN packets.

If a USB *peripheral* wishes to configure an endpoint to *send* data, it configures the endpoint as *USB\_EPDIRN\_OUT*. It responds to IN packets.

If a USB *peripheral* wishes to configure an endpoint to *receive* data, it configures the endpoint as *USB\_EPDIRN\_IN*. It responds to OUT packets.

By specifying endpoints in terms of their absolute data direction, not their USB packet type, we do not have any confusion when the device is operating in the OTG mode where it can change between host and peripheral when using the same USB peripheral.

## 2.12 Memory Configuration

When the USB core is enabled, the top 4Kbytes of memory from the internal eCOG1X RAM is assigned to the USB core and is used for the input and output buffers for endpoints EP1-3. This memory can no longer be accessed directly by the eCOG1X CPU.

The start addresses for these endpoint buffers within this reserved memory area are set by the **ep\*in\_addr** and **ep\*out\_addr** registers. The sizes of these buffers are fixed. The USB library sets up these buffers with suitable start addresses, and normally it is not necessary to change them. The memory map set up by the library for these buffers is shown below.

|          | Address range   |
|----------|-----------------|
| Unused   | 0x0E80 – 0x0FFF |
| EP3 OUT  | 0x0A80 – 0x0E7F |
| EP2 OUT  | 0x0880 – 0x0A7F |
| EP1 OUT  | 0x0780 – 0x087F |
| EP3 IN   | 0x0380 – 0x077F |
| EP2 IN   | 0x0180 – 0x037F |
| EP1 IN   | 0x0080 – 0x017F |
| Reserved | 0x0000 – 0x007F |

**Figure 4: Endpoint buffer memory map**

Note that these start addresses are relative to the base of this block of internal memory that is assigned to the USB core, not to the base address of the physical internal memory.

## 2.13 Background Timer

The USB library uses the eCOG1X timer/counter CNT1 as a background timer. This timer ensures that no part of a USB transfer can stall and stop the application from running. In order to use this timer, ensure that the constant `USB_USE_TIMER` is defined and that the following entry is present in the interrupt vector table:

```
cnt1_exp: usb_interrupt_timer
```

If this timer is unavailable, care must be taken when using the USB library that it does not get into a stalled state. At the least, the eCOG1X watchdog timer can be used to force a device reset.

## 3 Writing Software

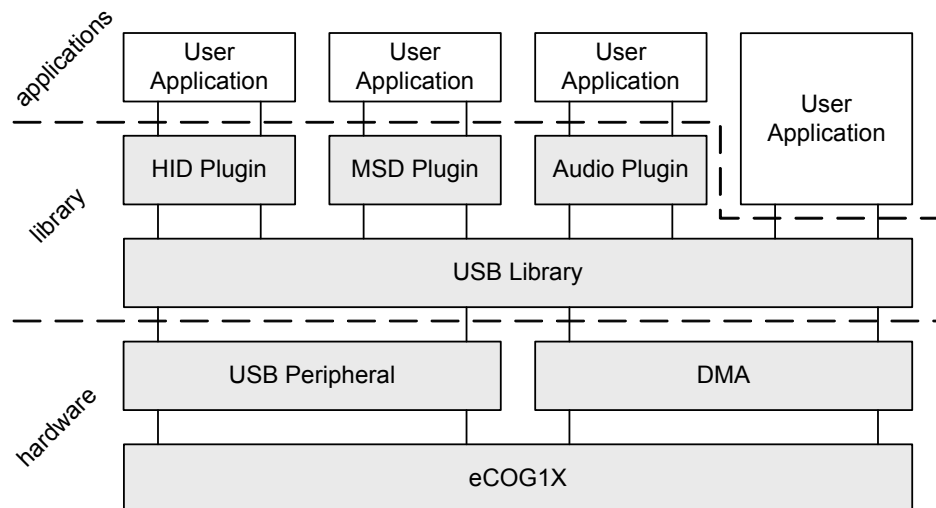
### 3.1 The Library Model

The library is written to provide an event driven messaging model, which a developer can build on by responding to the events raised during USB operation. The event model offers great flexibility, as a developer can respond to as many or as few events as they require. Many of the common events are automatically handled by the library which means that often only small additions are needed to provide custom functionality over and above standard behaviour.

For rapid development, a developer can use one of the provided *plugins* to the library, which provides additional support for common USB applications. Plugins are provided for keyboard, mass storage device and audio device and this range can be extended in the future. Developers who simply wish to emulate a common USB device can simply compile in the appropriate plugin code (see section 4) and, with minimal configuration, provide a complete USB solution.

For extreme examples, for example time-critical transfers or minimal code footprint, a developer can choose to exclude the event handling that the library provides and control the USB core directly. The library provides all the constant values needed to control the USB core registers as well as helper functions for common operations.

The figure below shows how applications can be developed with the library architecture:



**Figure 5: USB library architecture**

In summary, the possible ways of using the library are listed here, with the simplest first:

1. Use a library plugin to provide the functionality of a common USB device and only write the configuration code needed (see section 4).
2. Use the library event handler to manage many of the common events and write additional code to handle the extra required events.
3. Control the USB core directly, either by writing interrupt handlers or polling for events.

## 3.2 Including The USB Library In A Project

In order to access the USB library from a CyanIDE project, the following header file needs to be included in the source files:

```
#include <usb_lib.h>
```

Select **Project->Properties** from the CyanIDE main menu. Add the path:

```
$CYANIDE_INSTALLDIR\libraries\eCOG1X\USB
```

to both the *Additional Include Directories* item under *Compiler->Directories* and to the *Additional Library Directories* item under *Linker->Directories*. In both path lists, separate multiple paths with semi-colons.

## 3.3 Library Plugins

In order to reduce the amount of work required by a developer to add common USB functionality to their code, a number of plugins are provided which can be used unmodified to increase the application's functionality. All the developer needs to do is to configure the plugin for their particular requirements.

For example, the Mass Storage Device (MSD) plugin adds all of the code required to behave as a large virtual read-only FAT32 USB disk. A developer simply adds the following file to their CyanIDE project:

```
$CYANIDE_INSTALLDIR\libraries\eCOG1X\USB\usb_plugin_msd_peripheral.c
```

and then provides a number of callback functions that the plugin needs to operate. Prototypes for the functions implemented in the plugins provided with the library are defined in the include file *usb\_plugins.h*. An example of a typical callback function is:

```
void msd_get_drive_geometry(unsigned int *start_head,
    unsigned int *end_head, unsigned int *start_sector,
    unsigned int *end_sector, unsigned int *start_cylinder,
    unsigned int *end_cylinder);
```

which returns details of the virtual disk that is being simulated. Similar callback functions describe the arrangement of virtual files on the disk, and provide the contents of the files when requested by the host.

Plugins make behaving like common USB devices as simple as configuring them, hiding the lower level events and USB transactions from the developer. If a required plugin is not provided as part of the library, the developer must write their own event handling routines in order to provide the correct behaviour.

## 3.4 Adding USB Functionality To Existing Code

It is possible to use all of the programming methods described in section 3.1, both for new code and for adding USB functionality to existing code. For new code, the event model is the base of the application and the developer simply adds code to respond to the desired events.

For existing code, it is likely that there is already a main loop in the program and the USB library must be checked at intervals to see if there are any events pending. This is done through a call to the function `usb_check_for_event()`, which returns *TRUE* if an event needs to be handled, *FALSE* otherwise. Any pending interrupts can be handled by `usb_wait_for_event()` and `usb_event_host()` or `usb_event_peripheral()` as before. This allows the existing code to run as before when there are no USB events pending, and the library code is only called when needed.

### 3.5 DMA and FIFO Transfers

When using endpoints EP1-3, data can be transferred from the CPU to the USB core either by DMA or FIFO accesses. DMA offers the most flexibility, with transfers operating in the background while the CPU continues to execute instructions. Any address in internal memory can be used as the source for a DMA transfer; care must be taken during the DMA transfer that the source data is not modified. The DMA controller provides two channels, to support concurrent read and write transfers, with each channel containing a two transfer pipeline. The pipeline allows a constant stream of transfers to be queued for maximum performance.

The DMA peripheral works by driving the USB core slave FIFO. However, the FIFO can be accessed directly by the processor if required. The following table shows the relative advantages and disadvantages of using DMA over FIFO.

| FIFO   | DMA   |
|--|---|
| Endpoints 1 and 2 allow data to be double-buffered.                                  | Both DMA channels allow data to be pipelined when accessing the USB core.                                 |
| Data is copied into the FIFO before transmission.                                    | Data transfers are performed “in place”. The user must not modify the data while it is in use by the DMA. |
| FIFO returns an error if it is unable to accept data or there is no data to be read. | DMA stalls until the USB core is ready to accept or deliver data.   |
| Data must be accessed from the FIFO two or four bytes at a time.                     | Data can be byte aligned.   |
| Can copy data from any memory type.  | DMA can only transfer data to or from internal eCOG1X memory.   |

**Table 3: Comparison of FIFO and DMA transfers**

### 3.6 Configuring the eCOG1X for USB

The USB core is set up automatically by CyanIDE when the USB function is added to an eCOG1X device in the configuration editor. The setup code is kept in a function called `_usb_setup()` in the USB core library. This function is intended to be called by the CyanIDE generated code and not by the user application..



**Note: As a general rule, try to avoid enabling interrupts that are not required. Extra interrupts can slow down the processing of events in the event queue, especially for time-critical applications. If a particular interrupt is not required, it can always be disabled after setup with a statement such as:**

```
usb_fd->otg_cfg1_int_en.srpdet = 0;
```

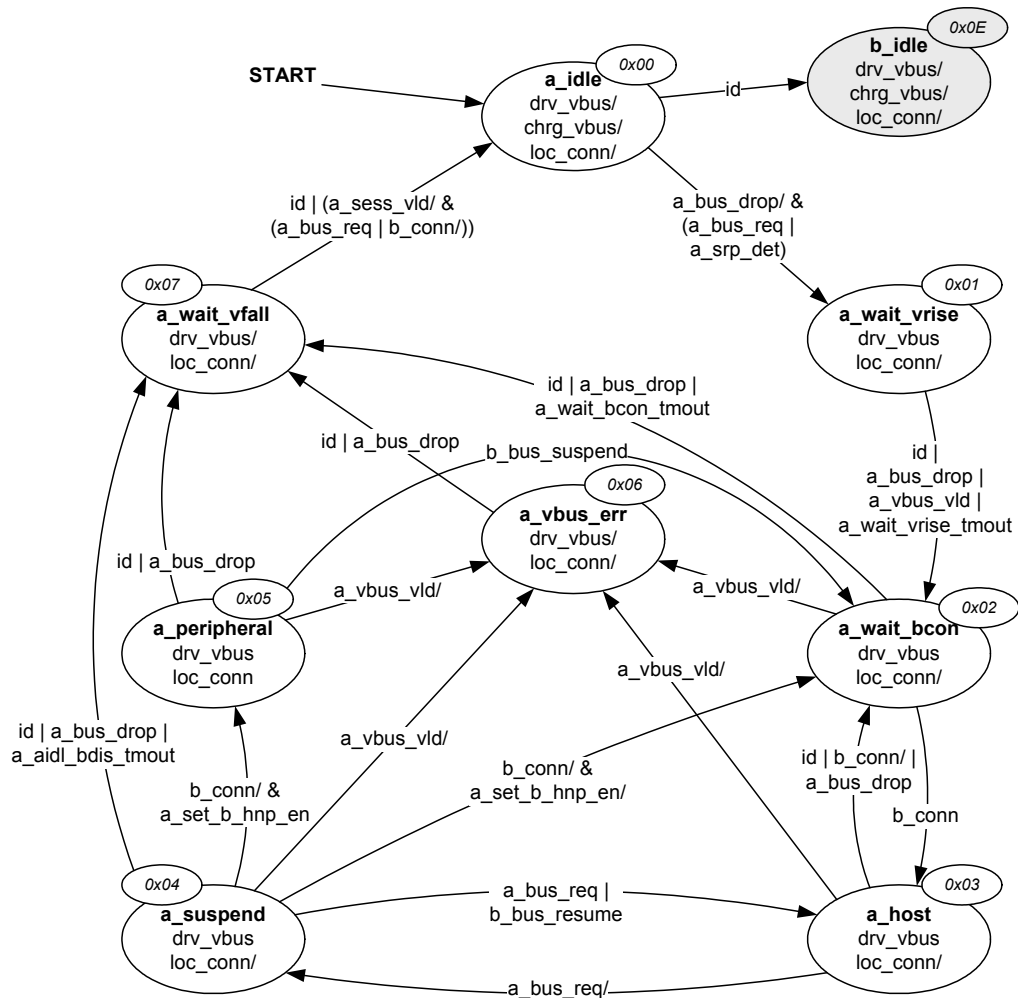
The USB core is now ready for use.

### 3.7 Host, Peripheral and OTG Operation

The eCOG1X supports use as a USB host, peripheral or On-The-Go (OTG). In all cases, the initial setup of the USB core is the same.

### 3.7.1 Host

In order to use the eCOG1X as a USB host, connect the USB\_OTG\_IDDIG pin to ground. This indicates to the USB core that it is a host device (A-device). The USB core uses the following state machine when operating as a host.



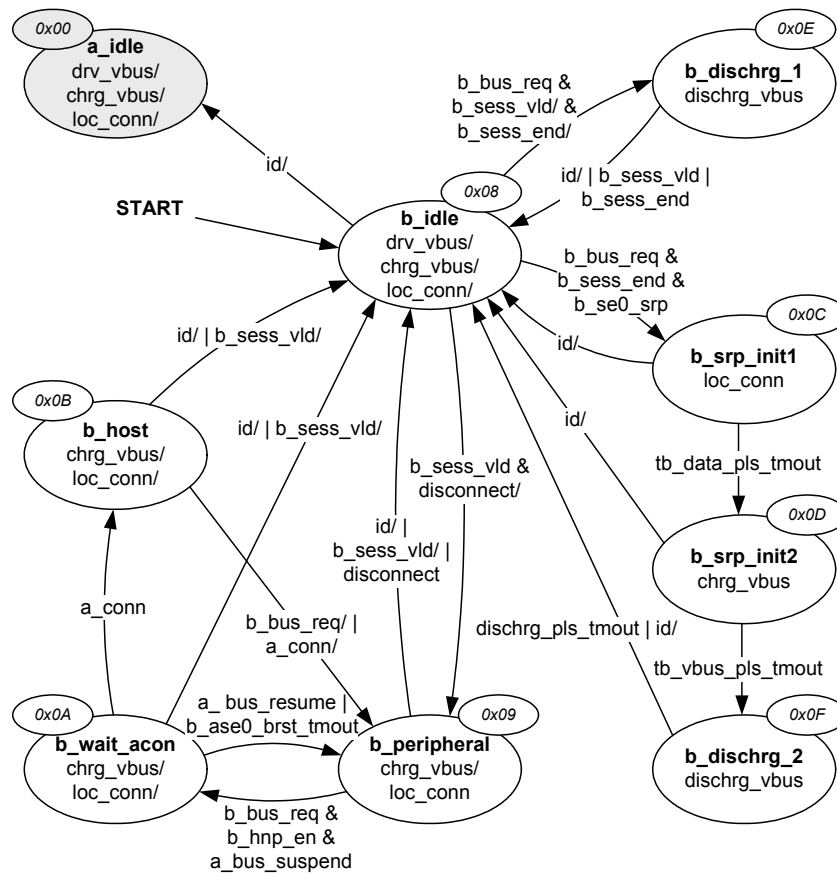
**Figure 6: A-device state machine**

This state machine is based on figure 6.2 in *On-The-Go Supplement to the USB 2.0 Specification, Revision 1.0a*.

With the ID pin grounded, the USB core cannot become a USB peripheral unless requested by an OTG device. If the application is not operating as an OTG device, return FALSE in the function `usb_plugin_allow_bhnp_enable()` in the plugin. This is done automatically in the example plugins.

### 3.7.2 Peripheral

In order to use the USB core as a USB peripheral, leave the USB\_OTG\_IDDIG input pin floating so that it is pulled up internally. The USB core uses the following state machine when operating as a peripheral device (B-device).



**Figure 7: B-device state machine**

This state machine is based on fig. 6.3 in *On-The-Go Supplement to the USB 2.0 Specification, Revision 1.0a*.

With the ID pin floating, the USB core cannot become a host unless it requests it using the OTG host negotiation protocol (see section 3.7.3).

### 3.7.3 OTG

The OTG standard requires that an OTG compatible device has an external 5 pin mini AB connector. The extra pin over a standard USB connector is provided as an ID pin. An OTG cable contains a plug at either end, one of which grounds the ID pin, and the other end leaves it floating. This determines the initial state of the USB core (host or peripheral). OTG uses both the state machines described in sections 3.7.1 and 3.7.2. The device that has its ID pin grounded is the "A-device", the other is the "B-device".

An OTG peripheral may request host status; that is, a B\_PERIPHERAL may issue a request to become a B\_HOST. The OTG host may allow the peripheral to become a host, and the A\_HOST then becomes an A\_PERIPHERAL.

For a full description of the host negotiation process, see *On-The-Go Supplement to the USB 2.0 Specification, Revision 1.0a*.

## 3.8 Interrupts and Events

During the use of the USB core, interrupts come from five sources:

- The USB core.
- The USB FIFOs.
- The DMA peripheral.
- USB wakeup.
- The background timer CNT1.

When using the USB library, the interrupts are handled by the following routines:

- `usb_interrupt_core()` in the file *usb\_interrupt\_core.c*.
- `usb_interrupt_fifo()` in the file *usb\_interrupt\_fifo.c*.
- `usb_interrupt_dma()` in the file *usb\_interrupt\_dma.c*.
- `usb_interrupt_wakeup()` in the file *usb\_interrupt\_wakeup.c*.
- `usb_interrupt_timer()` in the file *usb\_interrupt\_timer.c*.

which are referred to in the interrupt vector table in CyanIDE. The required source files for these interrupt handlers are provided in the USB core library and are linked in automatically when the project is built.

When using the USB library, it is not necessary to handle the USB interrupts directly.

For every interrupt, the USB library generates an *event*. An event is an 8 bit value that describes the interrupt that caused it, as well as any additional information that will help in the processing of the event. Events are used in preference to the raw interrupts, as the interrupts store their information in different ways, depending on the interrupt. The USB library translates this information into a single value, as well as clears the interrupt source cleanly for the next interrupt to occur.

Each interrupt handler provided by the USB library creates an event, which is placed in a queue when the interrupt occurs. The next event may be retrieved from the queue by calling `usb_wait_for_event()`, or the state of the queue may be queried by calling `usb_check_for_event()`.

The USB library queues and prioritises events to ensure that none are lost and that they are delivered in the optimum order for the developer. The size of the queue is given by the constant `USB_NUM_EVENTS`, which is defined in the include file *usb\_config.h*. By default, the queue stores 256 events. This value may be changed if required, but be aware that if it is necessary to increase this value, this is usually a sign that the events should be processed more quickly or that the number of interrupts in use should be reduced.



### 3.8.1 Special Events

Some events are given special treatment. Start-of-frame events (`USB_EVENTTYPE_SOF`) are not stored in the queue, as they can fill the queue very quickly if it is not continuously emptied. SOF events are used only for isochronous transfers, where it is not necessary to keep a history of all previous SOF events. All that is required is the time of the next SOF. When calling `usb_wait_for_event()`, an SOF event is returned if one is present, in preference to the normal event at the top of the queue.

Setup packet events are also given special treatment. Calling `usb_check_for_event()` with the parameter `USB_EVENTTYPE_SUDAV` returns TRUE if a SETUP packet has been read into the EP0 endpoint buffer. The `USB_EVENTTYPE_SUDAV` event retains its position in the event queue however, so a call to `usb_wait_for_event()` may not necessarily return the SUDAV event as the next event. This is to ensure that the events retain their basic chronological order (see section 3.8.2, Interrupt Priority).

SUDAV events are treated specially so that requests to stop isochronous transfers are not lost in the large number of SOF events. During an isochronous transfer, any SUDAV event should be processed in case it is a request to stop the isochronous transfer.

### 3.8.2 Interrupt Priority

The USB library interrupt service routines are designed to be very simple and to take very little time to complete. This means that the event queue almost always gives a true chronological representation of the order in which the events happened. However, it is possible for events be put into the queue out of chronological order in some situations.

One possible example is as follows. Consider sending isochronous data using DMA. All interrupts are turned on, meaning that interrupts are coming from the USB core, FIFOs and the DMA. At the end of the transfer, three interrupts fire in very close proximity:

1. The FIFO reports that it is empty.
2. The DMA reports that the transfer is complete.
3. The endpoint reports that the data has been sent.

While the first interrupt from the FIFO is being processed, interrupts 2 and 3 also fire. The eCOG1X does not have any interrupt priority scheme, so when the interrupt service routine ends, any new pending interrupts are dealt with in the order given in the interrupt vector table (see the eCOG1X User Manual, chapter 6). As the USB core has a higher priority in the interrupt vector table than the DMA transfer complete interrupt, the order of the events put in the event queue is:

1. FIFO.
2. Endpoint.
3. DMA.

Note how the interrupts 2 and 3 are swapped in the event queue.

The best way of dealing with this issue depends on the particular application. As a general rule, it is recommended that only the interrupts that are *necessary* to the application are enabled. All extra interrupts require extra processing and add extra complexity for the interrupt handling. In the above example, if all that is required is to check when the data is sent, then the FIFO and DMA interrupts may be disabled by calling the function `usb_core_interrupt_disable()`.

Alternatively, interrupts can be turned off individually. For example:

```
usb_fd->out_int_en.ep1 = 0;
```

This turns off the EP1 OUT interrupt that is generated every time a data packet is sent from the endpoint.

### 3.8.3 Adding User Events To The Queue

An event of type `USB_EVENTTYPE_USER` is reserved for use by application developers. Typically, a developer enters this event into the event queue when something important happens. The following requirement is important:

*Events can only be added to the event queue when the eCOG1X is in interrupt mode.*

Many events already naturally raise interrupts, so the event can be added to the queue at this time. Serial port activity, GPIO pins and timers can all generate suitable interrupts. If a user event needs to be added to the queue when not in interrupt mode (for example, when an ADC value is exceeded), an interrupt must be generated manually. One suggested method is to use an unused GPIO to generate an interrupt.

### 3.8.4 Event Types

The following event types are implemented:

| Event                                    | Description  |
|--|--|
| <code>USB_EVENTTYPE_BUSRESET</code>      | <i>Peripheral use:</i> The USB core generates this event when it detects start of a USB bus reset.<br><i>Host use:</i> USB reset interrupt request. The USB core generates this event when the host stops USB bus reset signalling.              |
| <code>USB_EVENTTYPE_EPnOUT</code>        | Data has been successfully sent on endpoint <i>EPn</i> .   |
| <code>USB_EVENTTYPE_EPnIN</code>         | Data has been successfully received on endpoint <i>EPn</i> .   |
| <code>USB_EVENTTYPE_EPnOUTERR</code>     | There has been a problem sending data on endpoint <i>EPn</i> .   |
| <code>USB_EVENTTYPE_EPnINERR</code>      | There has been a problem receiving data on endpoint <i>EPn</i> .   |
| <code>USB_EVENTTYPE_DMAdMEMWRAP</code>   | During either a DMA read or write operation on DMA channel <i>d</i> , the address pointer reference for the internal memory wrapped round a predefined memory boundary (indicating that the address of the intended transfer was out-of-bounds). |
| <code>USB_EVENTTYPE_DMAdRUF</code>       | During a read transfer from the USB core endpoint buffer to the internal memory using DMA channel <i>d</i> , the endpoint buffer emptied before the read transfer has completed.   |
| <code>USB_EVENTTYPE_DMAdWOF</code>       | During a write transfer from the internal memory the DMA channel <i>d</i> attempted to write more data than the USB core endpoint buffer can hold.   |
| <code>USB_EVENTTYPE_DMAdTRANSCMPL</code> | Indicates that a DMA transfer has completed on channel <i>d</i> and that the channel can be re-configured or re-armed ready to perform the next read or write transfer.  |
| <code>USB_EVENTTYPE_ISPERIPHERAL</code>  | The USB core has entered peripheral mode.  |
| <code>USB_EVENTTYPE_ISIDLE</code>        | The USB core has entered idle state.   |
| <code>USB_EVENTTYPE_SUDAV</code>         | Used in peripheral mode when a valid SETUP packet arrives.   |

**Table 4: Event types**

| Event                      | Description   |
|----------------------------|---|
| USB_EVENTTYPE_SRPDETECT    | Session Request Protocol detected. USB core must be in the idle state to generate this event.                         |
| USB_EVENTTYPE_ISHOST       | The USB core has entered host state.  |
| USB_EVENTTYPE_VBUSERR      | Generated in host mode if the Vbus voltage has dropped below acceptable levels.                                       |
| USB_EVENTTYPE_TIMER        | The USB timer has expired. This event is only generated if the constant USB_USE_TIMER is defined.                     |
| USB_EVENTTYPE_WAKEUP       | The USB core has detected a wakeup event.   |
| USB_EVENTTYPE_SUSPEND      | Used in peripheral mode when suspend signalling is detected.  |
| USB_EVENTTYPE_EPnFIFOEMPTY | Used in peripheral and host mode when the last byte is read from the EPn FIFO.  |
| USB_EVENTTYPE_EPnFIFOFULL  | Used in peripheral and host mode when the EPn FIFO buffer becomes full.   |
| USB_EVENTTYPE_SOF          | <i>Peripheral use:</i> Used when a Start-Of-Frame arrives from the host.<br><i>Host use:</i> Used when a SOF is sent. |
| USB_EVENTTYPE_SUTOK        | Used in peripheral mode when a valid SETUP token arrives.   |
| USB_EVENTTYPE_HIGHSPEED    | Used when the USB core enters high speed operation.   |
| USB_EVENTTYPE_EPnPING      | Used in host mode when a PING is received from the host on endpoint EPn.  |
| USB_EVENTTYPE_USER         | A user generated event has occurred.  |

Table 4: Event types

### 3.9 Configuring Endpoints

The default endpoint, EP0, is configured and ready to use as soon as the USB core is initialised. Additional endpoints, EP1 to EP3, must be configured before use. This is done with a call to `usb_epn_config()`, which takes the following parameters:

|                          |   |
|--------------------------|---|
| <b>local_endpoint</b>    | The local endpoint number on the eCOG1X: 1 to 3.  |
| <b>external_endpoint</b> | The endpoint number on the connected device.  |
| <b>channel</b>           | The DMA channel to use, either 0 or 1. Both DMA channels offer identical capabilities and DMA transfers can be used with endpoints EP1 to EP3. DMA cannot be used with the default endpoint EP0. Set the channel number to -1 if using FIFOs instead. |
| <b>direction</b>         | Sets the direction to be configured. Use <code>USB_EPDIRN_IN</code> for incoming data, <code>USB_EPDIRN_OUT</code> for outgoing data. See section 2.11 for a description of the endpoint directions used by the USB library.                          |
| <b>type</b>              | Use <code>USB_EP_TYPE_BULK_MASK</code> for bulk transfers, <code>USB_EP_TYPE_INTERRUPT_MASK</code> for interrupt transfers, and <code>USB_EP_TYPE_ISOCHRONOUS_MASK</code> for isochronous transfers.  |
| <b>buffering</b>         | Use <code>USB_EP_SINGLE_BUFFERED_MASK</code> for single buffered, and <code>USB_EP_DOUBLE_BUFFERED_MASK</code> for double-buffered. Only endpoints EP1 and EP2 support double buffering.  |
| <b>enable_endpoint</b>   | Leave the endpoint enabled after configuration.   |

Table 2 in section 2.6 shows a summary of the endpoints' capabilities.

### 3.10 Transferring Data Using The Default Endpoint, EP0

The method for using endpoint EP0 changes according to whether the device is configured as a USB host or a USB peripheral. As a host, EP0 is used to send data or request data from the connected peripheral. A peripheral responds to requests on EP0 by either reading the data sent by the host or sending data in response to the request from the host.

#### 3.10.1 General Data Transfer

Normal data is sent and received through endpoint EP0 using the functions `usb_ep0_write()` and `usb_ep0_read()`. These functions can be used by both USB host and peripheral devices.

Much of the data transferred over EP0 uses the SETUP packet format, and the USB library has additional functions to help deal with these packets.

#### 3.10.2 SETUP Packets - Host Behaviour

USB hosts use EP0 to send SETUP packets to the connected peripheral, which it does by using the `usb_ep0_write_setup()` function. The parameter for this function is a pointer to a structure of type `usb_setup_t`. The function can be used to send any valid SETUP packet.

The USB library also provides some additional functions which send preconfigured SETUP packets for common uses:

```
usb_get_descriptor_config();
    Returns the peripheral's configuration descriptor.

usb_get_descriptor_device();
    Returns the peripheral's device descriptor.

usb_set_address();
    Sets the address of the connected peripheral.

usb_set_config();
    Sets the peripheral into the desired configuration.

usb_hid_get_descriptor_report();
    Gets a report format from HID class peripherals.

usb_hid_set_descriptor_report();
    Sets the desired report format to use with HID class peripherals.

usb_hid_set_idle();
    Sets an interface idle time.

usb_hid_set_protocol();
    Sets the protocol to either boot or report.
```

#### 3.10.3 SETUP Packets - Peripheral Behaviour

After startup, a USB peripheral waits for SETUP packets from the host to arrive at EP0. These SETUP packets either send data to the peripheral, or request data from the peripheral.

When the host sends a SETUP packet to the peripheral, a `USB_EVENTTYPE_SUDAV` event occurs. The peripheral can then read the received data from the EP0 data area, using the function `usb_ep0_read_setup()`.

The function `usb_device_handle_req()` provides an example of how to handle a SETUP packet.

### 3.11 Transferring Data Using Endpoints EP1-EP3

Data can be read from and written to endpoints EP1 to EP3. Typically these endpoints are used to send large amounts of data, leaving EP0 for general USB configuration data. The USB core supports two methods for transferring data over EP1-EP3: FIFO or DMA. The relative merits of both methods is described in section 3.5.

The following functions are available for sending or receiving data over EP1-EP3:

```
usb_epn_write()  
usb_epn_read()
```

### 3.12 Suspending the USB Core

When the eCOG1X is working as a USB peripheral, the connected host may choose to suspend the peripheral to reduce power consumption and reduce the traffic on the USB bus. When the USB core is suspended, a `USB_EVENTTYPE_SUSPEND` event is generated; this is purely a notification event, as there is no further action required from the software to put the USB core into suspend mode.

When the host resumes operation, the USB core is woken automatically and events are generated as before. No specific event is generated when the USB core is resumed; any new event from the USB core indicates that it has come out of suspend state.

The USB library provides a function called `usb_is_suspended()` to check whether or not the USB core is suspended. It is important to check this before accessing the USB core registers. Access to the USB core registers generates a memory address exception if the USB core is suspended.



## 4 Plugins

### 4.1 What Plugins Offer

The USB library offers a set of functions that make controlling the USB core easy. It is likely that many applications of the eCOG1X will be very similar, for example, offering mass storage capability, a human interface device, or other common USB tasks. For this reason, several *plugins* are provided with the USB library that let developers plug-in additional functionality to the library to help with these common tasks. The plugin handles all the common USB traffic and only calls additional routines when it needs to read or write specific information.

For example, the USB peripheral keyboard plugin provides much of the functionality needed to send keystrokes to a USB host. All that is required is to provide three simple callback functions, which look like this:

```
unsigned int hid_get_vendor_id(void)
{
    return 0xABCD;
}

unsigned int hid_get_product_id(void)
{
    return 0x1234;
}

unsigned int hid_get_keystroke(unsigned int *modifiers)
{
    // Example: get ACSII character from the serial port
    char c = getchar();

    // Convert ASCII to keycode
    return usb_hid_ascii_to_keycode(c, modifiers);
}
```

These three functions provide the plugin with the custom data it needs to behave like a complete USB peripheral keyboard. In this case, there is no need for the developer to manually handle any of the USB events.

A developer may choose to write their own plugin to allow code re-use in several different projects. The advantage of plugins is that different functionality is achieved by modifying callback routines, not by copying and modifying the existing library code.

## 4.2 Using The Plugin Architecture

The plugin architecture builds on the basic event handling code in the USB library, as shown in Figure 5.

A typical `main()` function may contain the following code:

```
while (TRUE)
{
    // Wait for an interrupt
    eventType = usb_wait_for_event();

    // Pass on the event to the application code
    usb_event_peripheral(event_type);
}
```

The library function `usb_event_peripheral()` handles some of the common USB requests when working as a peripheral. For any application specific requests, these are passed to the plugin. If the plugin is added to the CyanIDE source tree, for example:

```
libraries\eCOG1X\USB\usb_plugin_keyboard_peripheral.c
```

then the plugin provides all the functionality of a basic USB keyboard. All that is required is to add the callback functions to return the USB Vendor ID, Product ID and keystroke. The *KeyboardPeripheral* example project shows how all these elements come together in a complete project.

In summary, the 3 basic steps when using a plugin are:

1. Call the USB default event handler (usually in `main()` function)
2. Add the desired plugin to the CyanIDE project
3. Provide any necessary callback functions to customise the behaviour



## 4.3 Keyboard Peripheral Plugin

### 4.3.1 Capabilities

The keyboard plugin offers the following functionality:

- Custom Vendor and Product ID
- Deliver USB keycodes on demand

Any project that uses this plugin must define the `USB_USEPERIPH_HID` constant in the Project Properties.

The plugin returns valid device, configuration, HID, interface and endpoint data when requested.

### 4.3.2 Callback Functions

The following callback functions are required:

---

#### **hid\_get\_vendor\_id**

```
unsigned int hid_get_vendor_id(void)
```

Returns the Vendor ID for the peripheral keyboard. This must be a unique ID that has been purchased by the developer and reserved for them by the USB Implementers' Forum.

---

#### **hid\_get\_product\_id**

```
unsigned int hid_get_product_id(void)
```

Returns the Product ID for the peripheral keyboard.

---

#### **hid\_get\_keystroke**

```
unsigned int hid_get_keystroke(unsigned int *modifiers)
```

Returns the keystroke, along with any modifiers. A table of valid modifiers is given in the library include file *usb\_descriptors\_hid.h*. This function is called by the plugin when a user interrupt happens (see section 3.8.3 for generating user interrupts).

---

#### **hid\_event\_handler**

```
BOOL hid_event_handler(unsigned int eventType)
```

Gives the user a chance to handle the current event, before the plugin. Returns TRUE if the event has been handled and the plugin should not process the event further, FALSE otherwise.

## 4.4 Mass Storage Device Peripheral Plugin

### 4.4.1 Capabilities

The MSD peripheral plugin provides the following functionality:

- Custom Vendor and Product ID.
- Text strings for manufacturer, product and serial number.
- Custom virtual drive geometry for a FAT32 read-only disk.
- Custom file structure, including nested directories.
- Short filename support only.

Any project that uses this plugin must define the `USB_USEPERIPH_MSD` constant in the Project Properties.

To use DMA transfers, uncomment the following line at the start of *plugin\_msd.c*:

```
#define USEDMA
```

If this line is commented out, then FIFO transfers are used instead.

The FAT file system is described in the following document, available from Microsoft:

Microsoft Extensible Firmware Initiative  
FAT32 File System Specification  
FAT: General Overview of On-Disk Format

### 4.4.2 Callback Functions

The following callback functions are required:

---

#### **msd\_get\_vendor\_id**

```
unsigned int msd_get_vendor_id(void)
```

Returns the Vendor ID for the MSD peripheral. This must be a unique ID that has been purchased by the developer and reserved for them by the USB Implementers' Forum.

---

#### **msd\_get\_product\_id**

```
unsigned int msd_get_product_id(void)
```

Returns the Product ID for the MSD peripheral.

---

#### **msd\_event\_handler**

```
BOOL msd_event_handler(unsigned int event_type)
```

Gives the user a chance to handle the current event, before the plugin. Returns TRUE if the event has been handled and the plugin should not process the event further, FALSE otherwise.

---

**msd\_get\_string**

```
unsigned int *msd_get_string(unsigned int index,  
    unsigned int lang_id, unsigned int *num_bytes)
```

Returns a pointer to a Unicode formatted string of a given index and language ID. A string must be returned if a non-zero index has been given in any of the USB descriptors that support text strings. Set *num\_bytes* to the length of the string in characters. Note that the length of the string in characters is half the value of the length in bytes, as the Unicode characters are stored in 16 bit values.

---

**msd\_get\_drive\_geometry**

```
void msd_get_drive_geometry(unsigned int *start_head,  
    unsigned int *end_head, unsigned int *start_sector,  
    unsigned int *end_sector, unsigned int *start_cylinder,  
    unsigned int *end_cylinder)
```

Returns the drive geometry of the drive being emulated. No checks are made by the plugin that the values are valid, this should be done by the application.

---

**msd\_get\_sector\_info**

```
void msd_get_sector_info(unsigned long int *relative_sector,  
    unsigned long int *total_sectors)
```

Returns sector information about the drive being emulated. No checks are made by the plugin that the values are valid, this should be done by the application.

---

**msd\_get\_hidden\_sec**

```
unsigned int msd_get_hidden_sec(void)
```

Returns the number of hidden sectors on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_rsvd\_sec\_cnt**

```
unsigned int msd_get_rsvd_sec_cnt(void)
```

Returns the number of reserved sectors on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_num\_fats**

```
unsigned int msd_get_num_fats(void)
```

Returns the number of file allocation tables on the disk. See the FAT file system documentation for a description of this value.

**msd\_get\_sec\_per\_clus**

```
unsigned int msd_get_sec_per_clus(void)
```

Returns the number of sectors per cluster on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_sec\_per\_trk**

```
unsigned int msd_get_sec_per_trk(void)
```

Returns the number of sectors per track on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_num\_heads**

```
unsigned int msd_get_num_heads(void)
```

Returns the number of heads on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_tot\_sec32**

```
unsigned int msd_get_tot_sec32(void)
```

Returns the total number of sectors on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_fat\_sz32**

```
unsigned long int msd_get_fat_sz32(void)
```

Returns the size of the file allocation table in sectors on the disk. See the FAT file system documentation for a description of this value.

---

**msd\_get\_root\_fsnode**

```
usb_msd_fsnode_t *msd_get_root_fsnode(void)
```

Returns the root file system node for the files on the virtual disk. The structure of the file system is described below in section 4.4.5.

---

**msd\_get\_file\_contents**

```
void msd_get_file_contents(char *block_buffer,  
                           usb_msd_fsnode_t *fsnode, unsigned long int block_num)
```

Returns the contents of block *block\_num* from the file *fsnode* into the provided block buffer. This function can be used to return the contents of different files by looking at which *fsnode* in the file system description is requested. See section 4.4.3 for details of how to describe the contents of the file system.

---

**msd\_get\_scsi\_num\_blocks**

```
unsigned long msd_get_scsi_num_blocks(void)
```

Returns the number of blocks available on the disk as part of the SCSI command set.

---

**msd\_get\_scsi\_drive\_info**

```
void msd_get_scsi_drive_info(unsigned int *sectors_per_track,  
    unsigned int *data_bytes_per_sector,  
    unsigned int *num_cylinders,  
    unsigned int *start_cyl_wr_precomp)
```

Returns disk information as part of the SCSI command set.

### 4.4.3 Describing the File System Contents

For a description of how the FAT32 file system stores directory information, see section 4.4.5. The plugin requires that a file system description is supplied that can be used to build a suitable FAT32 directory hierarchy, without needing to describe all the possible fields.

The plugin builds its directory structure from `usb_msd_fsnode_t` structures, which contain the following members:

| Member        | Description  |
|---------------|--|
| name          | A pointer to a 11 character filename for the file or directory. The format of this string is described below.  |
| attr          | Attributes associated with the file. The following values are valid:<br><div style="margin-left: 40px;"> <code>USB_MSD_FILEATTR_READ_ONLY</code><br/> <code>USB_MSD_FILEATTR_HIDDEN</code><br/> <code>USB_MSD_FILEATTR_SYSTEM</code><br/> <code>USB_MSD_FILEATTR_VOLUME_ID</code><br/> <code>USB_MSD_FILEATTR_DIRECTORY</code><br/> <code>USB_MSD_FILEATTR_ARCHIVE</code> </div> |
| size          | The size of the file in bytes. Set to zero for directory entries.  |
| directory     | Pointer to either an array of the directory's contents, or the parent directory. See below for details of how directories are described.   |
| start_cluster | The number of the first cluster on the disk that contains data for this file ( <i>fsnode</i> ). This value is calculated automatically by the plugin and initially should be set to zero.  |
| size_clusters | The size of the file ( <i>fsnode</i> ) data in clusters. This value is calculated automatically by the plugin and initially should be set to zero.   |

**Table 5: File system structure**

### 4.4.4 Filename Format

Filenames are described as strings of 11 characters in length, with an additional terminating zero '\0' character. The first 8 characters are used as the filename, the last 3 characters are the filename extension. For example, a filename of:

`"12345678ABC"`

is displayed as

`12345678.ABC`

See the FAT file system documentation for a description of valid and invalid filenames and extensions.

#### 4.4.5 How Directories Are Stored

The root directory of the volume starts with the following `usb_msd_fsnode_t` structure:

| Member        | Description                |
|---------------|----------------------------|
| name          | "VOLUME_NAME"              |
| attr          | USB_MSD_FILEATTR_VOLUME_ID |
| size          | 0                          |
| directory     | NULL                       |
| start_cluster | 0                          |
| size_clusters | 0                          |

**Table 6: Root directory structure**

Any directory is described as:

| Member        | Description   |
|---------------|---|
| name          | "DIRCTRY_NAME"  |
| attr          | USB_MSD_FILEATTR_DIRECTORY  |
| size          | 0   |
| directory     | A pointer to an array of <code>usb_msd_fsnode_t</code> structures which describe the contents of the directory. |
| start_cluster | 0   |
| size_clusters | 0   |

**Table 7: Directory structure**

The directory itself is an array of `usb_msd_fsnode_t` structures. The first and second structures within the array must be the entries for the current directory (".") and the parent directory (".."). The *directory* member in the structure for the current directory points to itself, and the *directory* member in the structure for the parent directory entry points to the parent directory.

Example current directory entry:

| Member        | Description                |
|---------------|----------------------------|
| name          | USB_MSD_FILENAME_DOT       |
| attr          | USB_MSD_FILEATTR_DIRECTORY |
| size          | 0                          |
| directory     | A pointer to itself.       |
| start_cluster | 0                          |
| size_clusters | 0                          |

**Table 8: Current directory structure**

Example parent directory entry:

| Member        | Description                        |
|---------------|------------------------------------|
| name          | USB_MSD_FILENAME_DOTDOT            |
| attr          | USB_MSD_FILEATTR_DIRECTORY         |
| size          | 0                                  |
| directory     | A pointer to the parent directory. |
| start_cluster | 0                                  |
| size_clusters | 0                                  |

**Table 9: Parent directory structure**

## 4.5 Audio Device Peripheral Plugin

### 4.5.1 Capabilities

The audio plugin provides the following functionality:

- Custom Vendor and Product ID.
- Text strings for manufacturer, product and serial number.
- Audio input.
- Audio output.
- Ability to have audio output or input only.
- Configurable audio sample rate, sample size and number of channels.

Any project that uses this plugin must define the `USB_USEPERIPH_AUDIO` constant in the Project Properties.

To use DMA transfers, uncomment the following line at the start of the source file *usb\_plugin\_audio\_periheral.c*:

```
#define USEDMA
```

If this line is commented out, FIFO transfers are used instead.

### 4.5.2 Callback Functions

The following callback functions are required:

---

#### **audio\_get\_vendor\_id**

```
unsigned int audio_get_vendor_id(void)
```

Returns the Vendor ID for the audio peripheral. This must be a unique ID that has been purchased by the developer and reserved for them by the USB Implementers' Forum.

---

#### **audio\_get\_product\_id**

```
unsigned int audio_get_product_id(void)
```

Returns the Product ID for the audio peripheral.

---

#### **audio\_event\_handler**

```
BOOL audio_event_handler(unsigned int event_type)
```

Gives the user chance to handle the current event, before the plugin. Returns TRUE if the event has been handled and the plugin should not process the event further, FALSE otherwise.

---

#### **audio\_get\_string**

```
unsigned int *audio_get_string(unsigned int index,  
                               unsigned int lang_id, unsigned int *num_bytes)
```



Returns a pointer to a Unicode formatted string of a given index and language ID. A string must be returned if a non-zero index has been given in any of the USB descriptors that support text strings. Set *num\_bytes* to the length of the string in characters. Note that the length of the string in characters is half the value of the length in bytes, as the Unicode characters are stored in 16 bit values.

---

**audio\_get\_config**

```
void audio_get_config(BOOL* has_line_in, BOOL* has_line_out)
```

Set *has\_line\_in* to TRUE if the soundcard supports line-in. Set *has\_line\_out* to TRUE if the soundcard supports line-out. Both can be set to be TRUE, but if both are FALSE then the result is undefined.

---

**audio\_get\_sample\_info**

```
void audio_get_sample_info(unsigned long int* sample_rate,  
    unsigned int* sample_size_bits,  
    unsigned int* sample_num_channels,  
    unsigned int* max_packet_size)
```

Returns the format for the audio samples:

***sample\_rate***

The number of samples per second.

***sample\_size\_bits***

The sample size in bits (for example 16 bits).

***sample\_num\_channels***

The number of audio channels. Only values '1' (mono) and '2' (stereo) are supported.

***max\_packet\_size***

The maximum size of data packet that is sent by the endpoint. This value is in bytes, and must have the minimum value of *sample\_rate* x *sample\_num\_channels* x *sample\_size\_bits* / 8, if *sample\_size\_bits* is a multiple of 8. It is advisable to use a slightly larger value for this buffer, to cope with situations where each (micro)frame contains a slightly different number of audio samples due to quantisation.

---

**audio\_get\_samples**

```
void *audio_get_samples(unsigned int *audioSamplesSize)
```

Returns a pointer to the outgoing audio sample buffer and the size of the buffer in bytes.

---

**audio\_set\_samples**

```
void *audio_set_samples(unsigned int audioSamplesSize)
```

Returns a pointer to the incoming audio sample buffer. The number of bytes that will be written is given in *audioSampleSize*. This buffer must be at least *bMaxPacketSize* bytes in size to contain the received audio data.

## 4.6 Keyboard Host Plugin

### 4.6.1 Capabilities

The keyboard plugin provides the following functionality:

- Accepts any keyboard that has boot support
- Reads keystrokes from keyboard

Any project that uses this plugin must define the `USB_USEPERIPH_HID` constant in the Project Properties.

### 4.6.2 Callback Functions

The following callback functions are required:

---

#### **hid\_set\_keystroke**

```
void hid_set_keystroke(unsigned int modifiers, unsigned int keycode)
```

Called by the plugin to register another keystroke from the keyboard. Any modifiers are also passed; valid values are defined in the file *usb\_descriptors\_hid.h* in the USB library directory.

## 4.7 Mass Storage Device Host Plugin

### 4.7.1 Capabilities

The MSD host plugin provides the following functionality:

- Read and write to USB mass storage devices.
- Returns basic drive information.
- Block read/write capability.
- Connect/disconnect notification.

### 4.7.2 Callback Functions

The following callback functions are required:

---

#### **msd\_connect**

```
unsigned int msd_connect(void)
```

Called by the plugin to notify the application that a mass storage device has been connected to the host. The application may now read the drive information using the `msd_get_drive_info()` function, or perform block read and write.

---

#### **msd\_disconnect**

```
unsigned int msd_disconnect(void)
```

Called by the plugin to notify the application that a USB device has been disconnected from the host.

### 4.7.3 Plugin Functions

The following plugin functions are provided:

---

#### **msd\_get\_drive\_info**

```
usb_msd_drive_info_t *msd_get_drive_info(void)
```

Returns a pointer to a `usb_msd_drive_info_t` structure:

```
typedef struct usb_msd_drive_info
{
    usb_msd_scsi_cmd_inquiry_std_t cmd_inquiry_std;
    usb_msd_scsi_capacity_list_header_t capacity_list_header;
    usb_msd_scsi_capacity_list_desc_t capacity_list_desc;
    BOOL write_protect;
} usb_msd_drive_info_t;
```

This structure contains the connected mass storage device's response to `INQUIRY` and `READ_FORMAT_CAPACITIES` requests, as well as whether the device is write protected.

---

#### **msd\_read\_block**

```
BOOL msd_read_block(unsigned int *block_buffer,
    unsigned long block_index)
```

Reads a block from the connected mass storage device. The block buffer must be large enough to read the block. The block size is given in the `blockLengthH` (high byte) and `blockLengthL` (low byte) of the `usb_msd_scsi_capacity_list_desc_t` structure of the drive info.

Returns `TRUE` on success, `FALSE` otherwise.

---

#### **msd\_write\_block**

```
BOOL msd_write_block(unsigned int *block_buffer,
    unsigned long block_index)
```

Writes a block to the connected mass storage device. The block buffer must be large enough to read the block. The block size is given in the `blockLengthH` (high byte) and `blockLengthL` (low byte) of the `usb_msd_scsi_capacity_list_desc_t` structure of the drive info.

Returns `TRUE` on success, `FALSE` otherwise.

## 5 Examples

### 5.1 Peripheral Examples

#### 5.1.1 Keyboard

The peripheral keyboard example shows taking ASCII characters from the serial port and simulating keyboard keystrokes to a USB host.

The example demonstrates the following:

- Using the peripheral keyboard library plugin to implement most of the required functionality.
- Using the default control endpoint (EP0).
- Using an interrupt endpoint (EP1) for data output.
- Using DMA to perform data transfer.
- Generating a user event (`USB_EVENTTYPE_USER`).
- Converting ASCII codes to USB keystrokes.

The example uses the peripheral keyboard plugin (see section 4.3) to perform most of the USB event handling.

#### 5.1.2 Mass Storage Device

The MSD peripheral example shows working as a large (256MB) read-only mass storage device. This can be used as a method for transferring large amounts of generated or captured data from the eCOG1X.

This example demonstrates the following:

- Using the peripheral MSD library plugin to implement most of the required functionality.
- Using the default control endpoint (EP0).
- Using bulk endpoints (EP1, EP2) for data output and input.
- Using DMA or FIFO to perform data transfer.
- Returning text strings on request.
- Implementing a read-only FAT32 file system.

The example uses the peripheral MSD plugin (see section 4.4) to perform most of the USB event handling.

### 5.1.3 Audio Device

The audio peripheral example shows the eCOG1X working as a pseudo-soundcard with line-in and line-out connections. When the host requests audio from line-in, the eCOG1X provides a 1kHz sine wave sampled at 44.1kHz, 16 bit stereo. Audio sent from the host is displayed via the serial port as a simple text VU meter, showing the audio level.

This example demonstrates the following:

- Using the peripheral audio library plugin to implement most of the required functionality.
- Using the default control endpoint (EP0).
- Using isochronous endpoint (EP2) for both data input and output.
- Using DMA or FIFO to perform data transfer.
- Returning text strings on request.

The example uses the peripheral audio plugin (see section 4.5) to perform most of the USB event handling.

## 5.2 Host Examples

### 5.2.1 Keyboard

The host keyboard example shows the eCOG1X working as a USB host which can accept keystrokes from most USB keyboards.

This example demonstrates the following:

- Using the host keyboard library plugin to implement most of the required functionality.
- Using the default control endpoint (EP0).
- Using interrupt endpoint (EP1) for data input.
- Using DMA to perform data transfer.
- Detecting a suitable peripheral keyboard with boot protocol support.

The example uses the host keyboard plugin (see section 4.6). The connected USB keyboard must support the boot protocol, as the host keyboard plugin does not support the parsing of arbitrary HID reports.

### 5.2.2 Mass Storage Device

The host mass storage example shows the eCOG1X working as a USB host which can perform block reads from and writes to a connected USB flash drive.

This example demonstrates the following:

- Using the host mass storage device plugin to implement most of the required functionality.
- Using the default control endpoint (EP0).
- Using bulk endpoint (EP3) for data send and receive.
- Mapping different host endpoint numbers to connected peripheral endpoints.
- Detecting a suitable USB flash drive peripheral.
- Sending and receiving SCSI commands over USB.

The example code performs a sample block read from and write to block 1 when a suitable USB flash drive is connected.





## 6 Library Functions

This section provides details for each of the USB library functions.

### 6.1 Library Utilities

---

#### **usb\_get\_otgstate**

```
#include <usb_lib.h>
unsigned int usb_get_otgstate(void)
```

This function returns the state of the USB core OTG state machine. It returns one of the following values.

|                            |
|----------------------------|
| USB_OTG_STATE_A_IDLE       |
| USB_OTG_STATE_A_WAIT_VRISE |
| USB_OTG_STATE_A_WAIT_BCON  |
| USB_OTG_STATE_A_HOST       |
| USB_OTG_STATE_A_SUSPEND    |
| USB_OTG_STATE_A_PERIPHERAL |
| USB_OTG_STATE_A_VBUS_ERR   |
| USB_OTG_STATE_A_WAIT_VFALL |
| USB_OTG_STATE_B_IDLE       |
| USB_OTG_STATE_B_PERIPHERAL |
| USB_OTG_STATE_B_WAIT_ACON  |
| USB_OTG_STATE_B_HOST       |
| USB_OTG_STATE_B_SRP_INIT1  |
| USB_OTG_STATE_B_SRP_INIT2  |
| USB_OTG_STATE_B_DISCHRG1   |
| USB_OTG_STATE_B_DISCHRG2   |

**Table 10: USB core OTG states**

---

#### **usb\_global\_init**

```
#include <usb_lib.h>
void usb_global_init(void)
```

This function sets up the global data for use and must be performed before any other USB\_lib functions are used. This function is called as part of `usb_setup()` and is not normally called manually.

---

#### **usb\_is\_bdevice**

```
#include <usb_lib.h>
BOOL usb_is_bdevice(void)
```

Returns TRUE if the USB core is operating as a B-device, FALSE otherwise (operating as an A-device).

**usb\_is\_suspended**

```
#include <usb_lib.h>
BOOL usb_is_suspended(void)
```

Returns TRUE if the USB core is currently suspended, FALSE otherwise. No accesses should be made to the USB core registers when it is suspended; see section 3.12 for more information.

---

**usb\_srp\_enable**

```
#include <usb_lib.h>
void usb_srp_enable(void)
```

Enables SRP support when operating as an OTG host. This allows the USB core to automatically respond to D+ or Vbus pulsing from the connected peripheral by turning on Vbus.

---

**usb\_srp\_disable**

```
#include <usb_lib.h>
void usb_srp_disable(void)
```

Disables SRP support when operating as an OTG host.

## 6.2 USB Setup

---

### **usb\_bus\_reset**

```
#include <usb_lib.h>
void usb_bus_reset(unsigned int duration)
```

Performs a bus reset of length `duration`. The following values are valid durations:

| <b>duration</b> | <b>Duration</b>   |
|-----------------|-------------------|
| USB_RESET_10MS  | 10ms (test only)  |
| USB_RESET_55MS  | 55ms              |
| USB_RESET_1_6MS | 1.6ms (test only) |

**Table 11: USB bus reset duration values**

---

### **usb\_enable\_remote\_wakeup**

```
#include <usb_lib.h>
void usb_enable_remote_wakeup(BOOL enable)
```

Sets whether the USB peripheral supports remote wakeup. Remote wakeup is disabled by default. Set `enable` to TRUE to allow the USB host to remotely wake the device.

---

### **usb\_is\_remote\_wakeup\_enabled**

```
#include <usb_lib.h>
BOOL usb_is_remote_wakeup_enabled(void)
```

Returns TRUE if the USB peripheral supports remote wakeup, FALSE otherwise. See `usb_enable_remote_wakeup()` for details on enabling and disabling this function.

---

### **\_usb\_setup**

```
#include <usb_lib.h>
void _usb_setup(unsigned int mode)
```

Setup routine called by CyanIDE generated code. There should be no need to call this function directly from the user application.

---

### **usb\_suspend**

```
#include <usb_lib.h>
void usb_suspend(void)
```

Suspends the USB core and waits for one of the following events to wake it up:

1. Data line pulsing.
2. Vbus pulsing.
3. Setting the ***usb.mode.resume*** bit in the eCOG1X registers.

No accesses to the USB core registers should be performed while the core is suspended.

## 6.3 Default Endpoint Data

---

### **usb\_ep0\_read**

```
#include <usb_lib.h>
int usb_ep0_read(unsigned int* buffer, unsigned int buffer_size)
```

Reads data from IN packets into the supplied buffer. This function continues to read packets until one of the following conditions occurs:

1. The supplied buffer is full.
2. A short data packet is read in.

Returns the number of bytes read, or -1 on any error.

---

### **usb\_ep0\_read\_setup**

```
#include <usb_lib.h>
void usb_ep0_read_setup(usb_setup_t* setup)
```

Reads the setup packet from the EP0 data buffer into the supplied request structure. Rearms EP0 so that it is ready to receive more data.

---

### **usb\_ep0\_write**

```
#include <usb_lib.h>
int usb_ep0_write(unsigned int* buffer, unsigned int buffer_size,
    unsigned int w_length)
```

Writes data from the supplied buffer to EP0 OUT packets. The way the data is written will depend on the values of *buffer\_size* and *w\_length*. The parameter *buffer\_size* contains the size of the supplied buffer in bytes, and *w\_length* is the amount of data requested (in bytes).

If *w\_length* <= *buffer\_size*, the transmitted data is never terminated with a zero length data packet.

If *w\_length* > *buffer\_size*, the entire buffer is transmitted. A zero length data packet is added if the last packet size is not a short packet.

Returns the number of bytes written, or -1 on any error.

---

### **usb\_ep0\_write\_descriptors**

```
#include <usb_lib.h>
int usb_ep0_write_descriptors(void *desc_config_array[],
    unsigned int w_length)
```

Writes the array of configuration descriptors to EP0. The parameter *desc\_config\_array* is a NULL terminated array of pointers to configuration descriptor structures. The function writes up to *w\_length* bytes, terminating the transfer with a zero length data packet if necessary.

The size of the transmitted configuration descriptor structure is calculated automatically from the contents of the configuration descriptor array.

Returns the number of bytes written, or -1 on any error.

---

**usb\_ep0\_write\_setup**

```
#include <usb_lib.h>
int usb_ep0_write_setup(usb_setup_t* setup, unsigned int* buffer,
    unsigned int buffer_size)
```

Sends a SETUP packet out from EP0. Additional data can be sent by setting the buffer and *buffer\_size* values. If data is returned as part of the request, this is written to *buffer*, if supplied. If no additional data is to be transferred, set *buffer* to NULL.

Returns the number of bytes written, or -1 on any error.

## 6.4 Endpoints 1-3 Data

### usb\_epn\_config

```
#include <usb_lib.h>
```

```
void usb_epn_config(unsigned int local_endpoint,
    unsigned int external_endpoint, int channel,
    unsigned int direction, unsigned int type,
    unsigned int buffering, BOOL enable_endpoint)
```

Configures endpoints 1-3 ready for use. Use the following values:

| Parameter         | Value                                 | Description  |
|-------------------|---------------------------------------|--|
| local_endpoint    | 1-3                                   | Local endpoint number to configure on the eCOG1X.    |
| external_endpoint | 1-3                                   | Endpoint number on the connected USB device.         |
| channel           | 0 or 1 for DMA use<br>-1 for FIFO use | Select DMA channel.<br>Set to -1 for FIFO transfers. |
| direction         | USB_EPDIRN_IN                         | Read data  |
|                   | USB_EPDIRN_OUT                        | Write data   |
| type              | USB_EP_TYPE_BULK_MASK                 | Bulk transfers                                       |
|                   | USB_EP_TYPE_INTERRUPT_MASK            | Interrupt transfers                                  |
|                   | USB_EP_TYPE_ISOCHRONOUS_MASK          | Isochronous transfers                                |
| buffering         | USB_EP_SINGLE_BUFFERED_MASK           | Single buffering                                     |
|                   | USB_EP_DOUBLE_BUFFERED_MASK           | Double buffering                                     |

**Table 12: EP1-3 configuration parameters**

### usb\_epn\_enable

```
#include <usb_lib.h>
```

```
void usb_epn_enable(unsigned int endpoint, unsigned int direction,
    BOOL enable)
```

Enables or disables an endpoint. Use the following values:

| Parameter | Value          | Description                   |
|-----------|----------------|-------------------------------|
| endpoint  | 1-3            | Endpoint number to configure. |
| direction | USB_EPDIRN_IN  | Read data                     |
|           | USB_EPDIRN_OUT | Write data                    |
| enable    | TRUE           | Enable endpoint               |
|           | FALSE          | Disable endpoint              |

**Table 13: EP1-3 enable parameters**

Set *enable* to TRUE to enable the endpoint, FALSE to disable it.

---

**usb\_epn\_read**

```
#include <usb_lib.h>
int usb_epn_read(unsigned int endpoint, int channel,
    unsigned int* buffer, unsigned int buffer_size)
```

General purpose routine to read data from an endpoint. If channel is -1, the function calls `usb_epn_read_dma()`, otherwise it calls `usb_epn_read_fifo()`.

---

**usb\_epn\_read\_arm**

```
#include <usb_lib.h>
BOOL usb_epn_read_arm(unsigned int endpoint, BOOL wait_for_data)
```

Enables an endpoint ready for reading. If the eCOG1X is being used as a peripheral, setting `wait_for_data` to TRUE sets the function to wait until data is received before returning.

---

**usb\_epn\_read\_dma**

```
#include <usb_lib.h>
int usb_epn_read_dma(unsigned int endpoint, unsigned int channel,
    unsigned int* buffer, unsigned int buffer_size)
```

Reads data from an endpoint, using DMA to transfer the data from the USB core to eCOG1X internal memory. The endpoint must be previously configured using `usb_epn_config()` and the DMA channel initialised with `usb_dma_config()`. The destination address in *buffer* must be located in eCOG1X internal memory.

This function continues to read packets until one of the following conditions occurs:

1. The supplied buffer is full.
2. A short data packet is read in.

Can be used for reading data from bulk, interrupt and isochronous endpoints.

Returns the number of bytes read, or -1 on any error.

---

**usb\_epn\_read\_fifo**

```
#include <usb_lib.h>
int usb_epn_read_fifo(unsigned int endpoint, unsigned int* buffer,
    unsigned int buffer_size)
```

Reads data from an endpoint, using the FIFOs to transfer the data from the USB core to eCOG1X internal memory. The endpoint must be previously configured using `usb_epn_config()`.

This function continues to read IN packets until one of the following conditions occurs:

1. The supplied buffer is full.
2. A short data packet is read in.

Can be used for reading data from bulk, interrupt and isochronous endpoints.

Returns the number of bytes read, or -1 on any error.

**usb\_epn\_reset**

```
#include <usb_lib.h>
void usb_epn_reset(unsigned int endpoint)
```

Resets an endpoint.

---

**usb\_epn\_write**

```
#include <usb_lib.h>
int usb_epn_write(unsigned int endpoint, int channel,
    unsigned int* buffer, unsigned int buffer_size,
    unsigned int w_length)
```

General purpose routine to write data to an endpoint. If channel is -1, the function calls `usb_epn_write_dma()`, otherwise it calls `usb_epn_write_fifo()`.

---

**usb\_epn\_write\_dma**

```
#include <usb_lib.h>
int usb_epn_write_dma(unsigned int endpoint, unsigned int channel,
    unsigned int* buffer, unsigned int buffer_size,
    unsigned int w_length)
```

Writes data to an endpoint, using DMA to transfer data to the USB core from eCOG1X internal memory. The endpoint must be previously configured using `usb_epn_config()` and the DMA channel initialised with `usb_dma_config()`. The source address in *buffer* must be located in eCOG1X internal memory.

The way that the data is written depends on the values of *buffer\_size* and *w\_length*. The parameter *buffer\_size* contains the size of the supplied buffer in bytes, and *w\_length* is the amount of data to be sent, also in bytes.

If *w\_length* <= *buffer\_size*, the transmitted data is never terminated with a zero length data packet.

If *w\_length* > *buffer\_size*, the entire buffer is transmitted. A zero length data packet is added if the last packet size is not a short packet.

Can be used for writing data to bulk, interrupt and isochronous endpoints.

Returns the number of bytes written, or -1 on any error.



---

**usb\_epn\_write\_fifo**

```
#include <usb_lib.h>
int usb_epn_write_fifo(unsigned int endpoint, unsigned int* buffer,
    unsigned int buffer_size, unsigned int w_length)
```

Writes data to an endpoint, using FIFOs to transfer data to the USB core from eCOG1X internal memory. The endpoint must be previously configured using `usb_epn_config()`.

The way that the data is written depends on the values of *buffer\_size* and *w\_length*. The parameter *buffer\_size* contains the size of the supplied buffer in bytes, and *w\_length* is the amount of data to be sent, also in bytes.

If *w\_length* <= *buffer\_size*, the transmitted data is never terminated with a zero length data packet.

If *w\_length* > *buffer\_size*, the entire buffer is transmitted. A zero length data packet is added if the last packet size is not a short packet.

Can be used for writing data to bulk, interrupt and isochronous endpoints.

Returns the number of bytes written, or -1 on any error.

---

**usb\_is\_epn\_enabled**

```
#include <usb_lib.h>
BOOL usb_is_epn_enabled(unsigned int endpoint,
    unsigned int direction)
```

Returns TRUE if the given endpoint and direction are enabled, FALSE otherwise.

## 6.5 DMA Data Transfers

### **usb\_dma\_config**

```
#include <usb_lib.h>
void usb_dma_config(unsigned int channel, unsigned int endpoint,
    unsigned int read_burst_size, BOOL is_read)
```

Sets up a DMA channel for transferring data to or from an endpoint. Two DMA channels are available. Use the following parameters:

| Parameter       | Value                                       | Description           |
|-----------------|---|-----------------------|
| channel         | 0-1   | DMA channel.          |
| endpoint        | 1-3   | Endpoint number.      |
| read_burst_size | USB_DMA_CFG_RD_BURST_SIZE_32<br>(see below) | Read data burst size. |
| is_read         | TRUE  | Set up to read data.  |
|                 | FALSE                                       | Set up to write data. |

**Table 14: DMA configuration parameters**

Set *read\_burst\_size* to be the number of bytes to read from or write to memory in a single operation, before bus arbitration may give up the memory bus for other parts of the eCOG1X to use. Larger values result in faster transfers, but potentially may cause problems with bus arbitration for other functions. The burst size can be set to 4, 8, 16, 32, 64 or 128 bytes. A value of zero in this field sets an unlimited burst size where the DMA transfer runs to completion before releasing the memory bus. A recommended value is 32 bytes, which gives good overall operation.

Set *is\_read* to TRUE if the DMA channel is to read from memory, FALSE if it is to write to memory.

This function must be called before any DMA transfers are performed, and is called as part of `usb_epn_config()`.

### **usb\_dma\_transfer**

```
#include <usb_lib.h>
void usb_dma_transfer(unsigned int channel, char* buffer,
    unsigned int buffer_size)
```

Perform a DMA transfer; the direction of the transfer is determined when the DMA channel is initialised with `usb_dma_config()`.

The source or destination address *buffer* is passed as a byte pointer. Transfers can start at any byte offset within the eCOG1X internal memory. Similarly, *buffer\_size* is specified in bytes and can be any size.

This function only begins the transfer, it does not wait until the transfer is complete.

## 6.6 Interrupts

### **usb\_core\_interrupt\_clear**

```
#include <usb_lib.h>
void usb_core_interrupt_clear(unsigned int mask);
```

Clears the following eCOG1X interrupts:

| mask                      | Interrupt   |
|---------------------------|---|
| USB_INTERRUPT_CORE_MASK   | Generated by the USB core. For more details on the cause of this interrupt, see the <i>int_num</i> member of the <i>usb_reg_t</i> structure, described in the USB Core User Manual. |
| USB_INTERRUPT_FIFO_MASK   | Generated by the USB core when using the onboard FIFO to transfer data.   |
| USB_INTERRUPT_WAKEUP_MASK | Generated by a USB wakeup event.  |
| USB_INTERRUPT_DMA_MASK    | Generated by the eCOG1X when performing endpoint data transfers using DMA.  |

**Table 15: USB core interrupt clear parameters**

### **usb\_core\_interrupt\_disable**

```
#include <usb_lib.h>
void usb_core_interrupt_disable(unsigned int mask);
```

Disables USB related interrupts. For a list of supported interrupts, see `usb_core_interrupt_clear()`.

### **usb\_core\_interrupt\_enable**

```
#include <usb_lib.h>
void usb_core_interrupt_enable(unsigned int mask);
```

Enables USB related interrupts. For a list of supported interrupts, see `usb_core_interrupt_clear()`.

### **usb\_interrupt\_core**

```
#include <usb_lib.h>
void __irq_entry usb_interrupt_core(void)
```

The default interrupt handler for the USB core. Read the events created by this function using `usb_wait_for_event()`.

### **usb\_interrupt\_dma**

```
#include <usb_lib.h>
void __irq_entry usb_interrupt_dma(void)
```

The default interrupt handler for the USB DMA channels. Read the events created by this function using `usb_wait_for_event()`.

**usb\_interrupt\_fifo**

```
#include <usb_lib.h>
void __irq_entry usbInterruptDMA(void)
```

The default interrupt handler for USB FIFO transfers. Read the events created by this function using `usb_wait_for_event()`.

---

**usb\_interrupt\_timer**

```
#include <usb_lib.h>
void __irq_entry usb_interrupt_timer(void)
```

The default interrupt handler for the USB background timer. Read the events created by this function using `usb_wait_for_event()`.

See section 2.13 for more details on the background timer.

---

**usb\_interrupt\_wakeup**

```
#include <usb_lib.h>
void __irq_entry usb_interrupt_wakeup(void)
```

The default interrupt handler for USB wakeup events. Read the events created by this function using `usb_wait_for_event()`.

## 6.7 Events

---

### **usb\_check\_for\_event**

```
#include <usb_lib.h>
BOOL usb_check_for_event(unsigned int event_to_check)
```

Checks to see whether a particular event has happened. It is not possible to check for all event types; the following events are supported:

| event_to_check            | Checks for...              |
|---------------------------|----------------------------|
| USB_EVENTTYPE_SUDAV       | Valid SETUP packet in EP0. |
| USB_EVENTTYPE_SOF         | Start of frame.            |
| USB_EVENTTYPE_UNSUPPORTED | Any event.                 |

**Table 16: Check event parameters**

Returns TRUE if the desired event has happened, FALSE otherwise.

This function does not remove events from the event queue. Typically it is used when some background processing is being performed; it allows the status of the event queue to be checked without changing it and to request events from the queue only when there are any present.

---

### **usb\_clear\_sof**

```
#include <usb_lib.h>
void usb_clear_sof(void)
```

Clears any pending USB\_EVENTTYPE\_SOF events from the event queue.

This function is used during isochronous transfers to ensure that data is sent or received in time with the USB frames. This function can be called after a long operation has completed, so that the data transfer begins at the start of the next frame, rather than immediately because a previous SOF is in the event queue.

See section 3.8.1 for details on how the SOF events are handled by the event queue.

---

### **usb\_event\_host**

```
#include <usb_lib.h>
void usb_event_host(unsigned int event_type)
```

Used when operating as a USB host. Call this function from the main event loop with the first event taken from the event queue (see section 3.8 for details on the event queue). This function handles some of the common events that are generated when working as a USB host. Any events that are not handled by this function are passed on to the plugin, see section 4.2.

**usb\_event\_peripheral**

```
#include <usb_lib.h>
void usb_event_peripheral(unsigned int event_type)
```

Used when operating as a USB peripheral. Call this function from the main event loop with the first event taken from the event queue (see section 3.8 for details on the event queue). This function handles some of the common events that are generated when working as a USB peripheral. Any events that are not handled by this function are passed on to the plugin, see section 4.2.

---

**usb\_timer\_start**

```
#include <usb_lib.h>
void usb_timer_start(unsigned int millisecs)
```

Starts the USB timer, which is used to ensure that `usb_wait_for_event()` never stalls entirely when no events are present. This function is called by `usb_wait_for_event()` and there should not be any need to call this function.

See section 3.8 for more information on avoiding stalls in the event queue.

---

**usb\_timer\_stop**

```
#include <usb_lib.h>
void usb_timer_stop()
```

Stops the USB timer, which is used to ensure that `usb_wait_for_event()` never stalls entirely when no events are present. This function is called by `usb_wait_for_event()` and there should not be any need to call this function.

See section 3.8 for more information on avoiding stalls in the event queue.

---

**usb\_wait\_for\_event**

```
#include <usb_lib.h>
unsigned int usb_wait_for_event(void)
```

Waits for the next event to fire, or takes one from the existing event queue. Returns the next event.

See section 3.8 for more information on the event queue.

## 6.8 Standard Requests

---

### **usb\_get\_descriptor**

```
#include <usb_lib.h>
BOOL usb_get_descriptor(unsigned int* dst_buffer,
    unsigned int dst_buffer_size, unsigned int* src_buffer,
    unsigned int src_buffer_size, unsigned int index,
    unsigned int descriptor_type)
```

Extracts descriptors from a buffer. Copies the descriptor at index if *descriptor\_type* is set to `USB_DESC_UNUSED`. Can be used to filter descriptors by setting *descriptor\_type* to the value of the desired type.

The source data buffer *src\_buffer* which contains all of the descriptors is obtained by calling `usb_get_descriptor_config()`.

Returns TRUE if a descriptor is found and returned, FALSE otherwise.

---

### **usb\_get\_descriptor\_config**

```
#include <usb_lib.h>
int usb_get_descriptor_config(unsigned int* buffer,
    unsigned int buffer_size)
```

Gets the configuration descriptor for the peripheral, as described in sec. 9.6.3 of the USB 2.0 specification. The descriptor is placed in the supplied buffer.

Use when operating as a host only.

Returns the number of bytes read, or -1 on any error.

---

### **usb\_get\_descriptor\_device**

```
#include <usb_lib.h>
int usb_get_descriptor_device(unsigned int* buffer,
    unsigned int buffer_size)
```

Gets the device descriptor for the peripheral, as described in sec. 9.6.1 of the USB 2.0 specification. The descriptor is placed in the supplied buffer.

Use when operating as a host only.

Returns the number of bytes read, or -1 on any error.

---

### **usb\_set\_address**

```
#include <usb_lib.h>
int usb_set_address(unsigned int address)
```

Sets the address of the connected peripheral, as described in sec. 9.4.6 of the USB 2.0 specification.

Use when operating as a host only.

Returns the number of bytes sent, or -1 on any error.

**usb\_set\_config**

```
#include <usb_lib.h>
int usb_set_config(unsigned int config)
```

Sets the configuration of the connected peripheral, as described in sec. 9.4.7 of the USB 2.0 specification.

Use when operating as a host only.

Returns the number of bytes sent, or -1 on any error.



## 6.9 Plugins

---

### **usb\_device\_handle\_setup**

```
#include <usb_lib.h>
BOOL usb_device_handle_setup(usb_setup_t* setup)
```

Called by `usb_event_peripheral()` to handle common requests. It is not necessary to call this function directly.

---

### **usb\_device\_handle\_setup\_audio**

```
#include <usb_lib.h>
BOOL usb_device_handle_setup_audio(usb_setup_t* setup)
```

Called by `usb_event_peripheral()` to handle common audio class requests. It is not necessary to call this function directly.

---

### **usb\_device\_handle\_setup\_hid**

```
#include <usb_lib.h>
BOOL usb_device_handle_setup_hid(usb_setup_t* setup)
```

Called by `usb_event_peripheral()` to handle common HID class requests. It is not necessary to call this function directly.

---

### **usb\_device\_handle\_setup\_msdc**

```
#include <usb_lib.h>
BOOL usb_device_handle_setup_msdc(usb_setup_t* setup)
```

Called by `usb_event_peripheral()` to handle common MSD class requests. It is not necessary to call this function directly.

## 6.10 MSD Class Specific Functions

---

### **usb\_msd\_get\_max\_lun**

```
#include <usb_lib.h>
```

```
int usb_get_max_lun(unsigned int interface)
```

Get the number of LUNs (Logical Unit Numbers) for the connected mass storage device.  
The number of LUNs indicates how many drives are present.

## 6.11 HID Class Specific Functions

---

### **usb\_hid\_ascii\_to\_keycode**

```
#include <usb_lib.h>
unsigned int usb_hid_ascii_to_keycode(char c,
    unsigned int *modifiers)
```

Returns a USB keycode from an ASCII value. Pass in `modifiers` to get any additional keys required to get the desired character. May return modifiers from the following:

| Modifiers             |
|-----------------------|
| USB_KEYMOD_LEFTCTRL   |
| USB_KEYMOD_LEFTSHIFT  |
| USB_KEYMOD_LEFTALT    |
| USB_KEYMOD_LEFTGUI    |
| USB_KEYMOD_RIGHTCTRL  |
| USB_KEYMOD_RIGHTSHIFT |
| USB_KEYMOD_RIGHTALT   |
| USB_KEYMOD_RIGHTGUI   |

**Table 17: Keycode modifiers**

See "USB HID Usage Tables, v.1.12", table 12 and "Device Class Definition for Human Interface Devices (HID) Version 1.11", section 8.3 for a full description of available modifiers and keystrokes.

---

### **usb\_hid\_get\_descriptor\_report**

```
#include <usb_lib.h>
int usb_hid_get_descriptor_report(unsigned int* buffer,
    unsigned int buffer_size)
```

Gets the HID report descriptor from the connected USB peripheral. The descriptor is placed in the supplied buffer.

Use when operating as a host only.

Returns the number of bytes read, -or 1 on any error.

---

### **usb\_hid\_keycode\_to\_ascii**

```
#include <usb_lib.h>
char usb_hid_keycode_to_ascii(unsigned int modifiers,
    unsigned int keycode)
```

Returns an ASCII character from a USB keycode with modifiers. Attempts to return the "best attempt" at a conversion, but may return '\0' if there is no suitable conversion.

For a table of accepted keycode modifiers, see `usb_hid_ascii_to_keycode()`.

**usb\_hid\_set\_descriptor\_report**

```
#include <usb_lib.h>
int usb_hid_set_descriptor_report(unsigned int interface,
    unsigned int report_id)
```

Sets the report on the given interface, as described in sec. 7.2.2 of the USB HID 1.11 specification.

Returns the number of bytes sent, -or 1 on any error.

Use when operating as a host only.

---

**usb\_hid\_set\_idle**

```
#include <usb_lib.h>
int usb_hid_set_idle(unsigned int interface)
```

Sends a *Set\_Idle* request to the peripheral interface, as described in sec. 7.2.4 of the USB HID 1.11 specification.

Returns the number of bytes sent, -or 1 on any error.

Use when operating as a host only.

---

**usb\_hid\_set\_protocol**

```
#include <usb_lib.h>
int usb_hid_set_protocol(unsigned int interface,
    unsigned int protocol)
```

Sends a *Set\_Protocol* request to the peripheral, as described in sec. 7.2.6 of the USB HID 1.11 specification.

Returns the number of bytes sent, -or 1 on any error.

Use when operating as a host only.

## 6.12 Helper Routines

---

### **usb\_print\_desc\_config**

```
#include <usb_lib.h>
void usb_print_desc_config(usb_desc_config_t* desc_config)
```

Prints the contents of a `usb_desc_config_t` structure. Useful for debugging purposes when developing operation as a USB host.

---

### **usb\_print\_desc\_device**

```
#include <usb_lib.h>
void usb_print_desc_device(usb_desc_device_t* desc_device)
```

Prints the contents of a `usb_desc_device_t` structure. Useful for debugging purposes when developing operation as a USB host.

---

### **usb\_print\_desc\_endpoint**

```
#include <usb_lib.h>
void usb_print_desc_endpoint(usb_desc_endpoint_t* desc_endpoint)
```

Prints the contents of a `usb_desc_endpoint_t` structure. Useful for debugging purposes when developing operation as a USB host.

---

### **usb\_print\_desc\_hid**

```
#include <usb_lib.h>
void usb_print_desc_hid(usb_desc_hid_t* desc_hid)
```

Prints the contents of a `usb_desc_hid_t` structure. Useful for debugging purposes when developing operation as a USB host.

---

### **usb\_print\_desc\_interface**

```
#include <usb_lib.h>
void usb_print_desc_interface(usb_desc_interface_t* desc_interface)
```

Prints the contents of a `usb_desc_interface_t` structure. Useful for debugging purposes when developing operation as a USB host.

---

### **usb\_print\_descriptors**

```
#include <usb_lib.h>
void usb_print_descriptors(unsigned int* buffer,
    unsigned int buffer_size)
```

Prints the descriptors contained in the buffer. Useful for debugging purposes when developing operation as a USB host.

**usb\_print\_descriptors\_hid**

```
#include <usb_lib.h>
void usb_print_descriptors_hid(unsigned int* buffer,
    unsigned int buffer_size)
```

Prints only the HID descriptors contained in the buffer. Useful for debugging purposes when developing operation as a USB HID host.

---

**usb\_print\_msdcbw\_header**

```
#include <usb_lib.h>
void usb_print_msdcbw_header(usb_msdcbw_header_t* cbw_header)
```

Prints the contents of a `usb_msdcbw_header_t` structure. Useful for debugging purposes when developing operation as a USB MSD peripheral.

---

**usb\_print\_msdsccsi\_cmd\_inquiry**

```
#include <usb_lib.h>
void usb_print_msdsccsi_cmd_inquiry(
    usb_msdsccsi_cmd_inquiry_t* cmd_inquiry)
```

Prints the contents of a `usb_msdsccsi_cmd_inquiry_t` structure. Useful for debugging purposes when developing operation as a USB MSD peripheral.

---

**usb\_print\_msdsccsi\_cmd\_read\_format\_capacities**

```
#include <usb_lib.h>
void usb_print_msdsccsi_cmd_read_format_capacities(
    usb_msdsccsi_cmd_read_format_capacities_t*
    cmd_read_format_capacities)
```

Prints the contents of a `usb_msdsccsi_cmd_read_format_capacities_t` structure. Useful for debugging purposes when developing operation as a USB MSD peripheral.

---

**usb\_assert\_brk**

`usb_assert_brk.asm`

This file provides an implementation of the `__assert()` function, as required by the USB library code. This file is added to the CyanIDE project automatically when a USB plugin is selected. It should not be necessary to add this file to the project manually.