



... the world's most energy friendly microcontrollers

EFM32 as USB Device

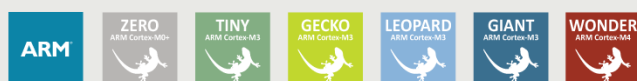
AN0065 - Application Note

Introduction

This application note introduces the EFM32 USB Device stack and explains how to configure the EFM32 to act as a USB Device.

This application note includes:

- This PDF document
- Source files (zip)
 - Example C-code
 - Multiple IDE projects



1 Introduction

Several EFM32 families include part numbers with integrated USB support. See the data sheet or product selector for information about which devices contain USB. The EFM32 USB can be operated in both host and device mode. This document discusses how to use the USB in *device* mode.

Some features of the EFM32 USB includes:

- Fully compliant with USB Specification, Revision 2.0
- Full-speed USB (12 Mb/s) host and device
- Support for Control, Bulk and Interrupt transfers
- Up to 12 configurable endpoints (6 IN, 6 OUT) in addition to EP0
- 2 kB of endpoint memory
- Low power mode with reset/resume detection

The EFM32 comes with a full USB Stack (software API). The USB stack is provided as C source files and full doxygen documentation is available.

1.1 More Information

In addition to this document, Silicon Labs provides several other resources on USB:

- emusb Doxygen Documentation
- AN0042 USB-UART Bootloader
- AN0046 USB Hardware Guidelines
- AN801 EFM32 as USB Host
- Kit examples
 - USB CDC Virtual COM port (usbdcdc)
 - USB HID Keyboard (usbdhidkbd)
 - USB Mass Storage Device (usbdmsd)
 - USB Vendor Unique Device (usbdvud)
 - USB Device Enumerator (usbhenum)
 - USB Host HID Keyboard (usbhhidkbd)
 - USB Host Mass Storage Device (usbhmsdfatcon)

The reader should also be familiar with the USB v2.0 Specification available from the Universal Serial Bus Implementers Forum (USB-IF).

2 Transfers

Communication over USB is done through logical channels called *pipes*. Each pipe specifies a transfer type, direction and is linked to an *endpoint* on the device.

2.1 Transfer Direction

In USB terminology data transfer direction is labelled *IN* if data flows from device to host or *OUT* if data flows from host to device.

2.2 Endpoints

Endpoints are entities on USB devices that facilitates communication. Each pipe is assigned to one (or two) endpoints. A pipe can be either unidirectional (one-way, most common) or bidirectional (two-way, used for control transfers). An endpoint is specified with both an endpoint number and a data flow direction (IN/OUT). Thus a bidirectional pipe (control pipe) uses one endpoint number with both directions, but two unidirectional pipes can use the same endpoint number if the direction is different.

Each endpoint specifies a size. This is the maximum amount of bytes that can be sent in one USB packet. On the EFM32, the maximum size of any endpoint is 64 bytes. Note that larger messages can be sent, but they will be split into several USB packets.

Endpoint zero (EP0) is always configured as a bidirectional control pipe (according to the USB spec). This endpoint is special because it handles both the device enumeration and configuration which is part of the USB specification. But it can also be extended to handle device specific configuration.

Figure 2.1. Endpoint addresses

Num	IN	OUT	Use
0	0x80	0x00	Control
1	0x81	0x01	Application defined
2	0x82	0x02	
3	0x83	0x03	
⋮	⋮	⋮	
6	0x86	0x06	

The *address* of each endpoint is described with an 8-bit value which incorporates both the endpoint number (bits [3:0]) as well as the transfer direction (bit 7, a '1' means IN). As an example, the endpoint number 2 with transfer direction IN will have address 0x82.

On the EFM32, up to six OUT and six IN endpoints can be configured, in addition to endpoint 0. Addresses/endpoint numbers do not need to be sequentially allocated.

2.3 Transfer Types

The USB specification defines four different transfer types:

- Control
- Interrupt
- Bulk
- Isonchronous

Of these, the EFM32 USB Stack supports the three first, but not isonchronous transfers.

2.3.1 Control Transfers

Control transfers are handled by bidirectional pipes. A control transfer is a 3-phase transaction. The first phase is always initiated by the host and contains a Setup packet. The Setup packet has a USB-defined structure and tells the device the type of transaction and direction. Then follows a data phase with zero or more data packets in the direction specified in the Setup packet. In the last phase the device sends a status message (ok/failed/wait).

Control transfers plays a special role in the USB specification because enumeration and configuration is done with control transfers. The EFM32 USB Stack handles the enumeration and basic configuration automatically, but the USB application can be forced to handle control transfers as well. For instance if the application is implementing a HID class, the application must respond to special HID control transfers.

The EFM32 USB Stack provides a callback function which can be used by the application to listen for control transfers. The `SetupCmd()` function is called whenever the USB receives a Setup packet on EP0. See the following kit examples for how to use this callback to listen for USB-defined and vendor-defined control messages:

- USB HID Keyboard Example (usbdhidkbd)
- USB Vendor Unique Device Example (usdbvud)

2.3.2 Interrupt Transfers

Interrupt transfers are used for transfers that need a bounded latency. Endpoints configured for interrupt transfers specify a maximum period between each time the transfer is serviced. Note that the actual latency can be longer than the configured value if there are errors in the transmission. In case of errors the transmission is retried, but this can cause a longer time until the message is delivered. Interrupt transfers are unidirectional.

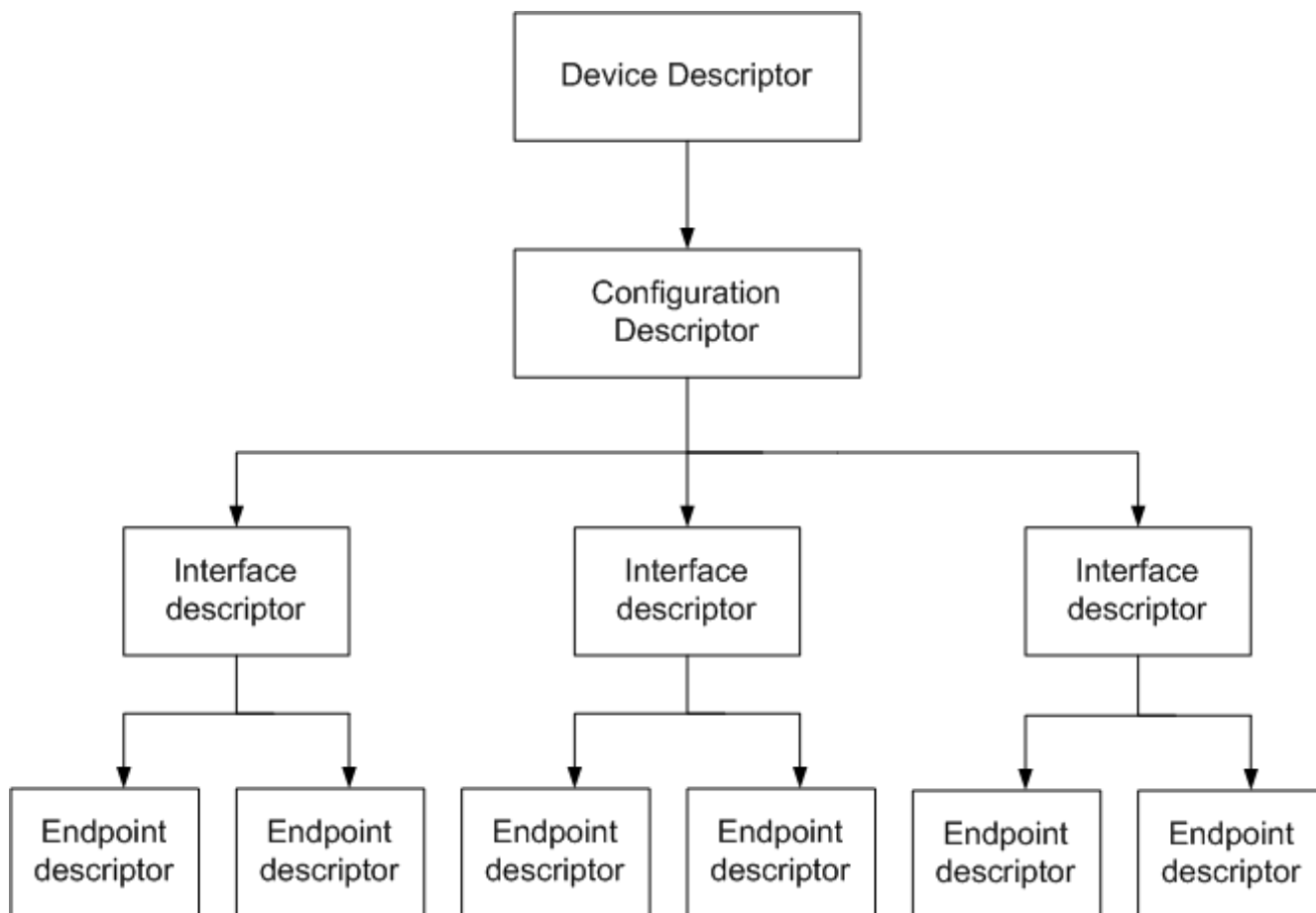
2.3.3 Bulk Transfers

Bulk transfers are intended for large amounts of data to be transferred when latency is not important. Bulk transfers will use all the available bandwidth on the bus. There is gurantee of delivery of the data by means of error checking and retrying, but there is no gurantee of latency. Bulk transfers are unidirectional.

3 Configuration

Each USB device must provide descriptors which provide the host with information about the device itself and the available features. The USB host uses the descriptors to identify the device, allocate bandwidth on the bus and configure each device. Descriptors are defined in the header file `descriptors.h`. Please refer to this file while reading this chapter. The USB descriptors are defined in a hierarchy, see Figure 3.1 (p. 5).

Figure 3.1. USB descriptor hierarchy



3.1 Device Descriptor

The device descriptor contains information about the device itself. This descriptor contains the Vendor ID (assigned by USB-IF) and Product ID (assigned by each manufacturer). A device can only have one device descriptor. The USB stack provides the `USB_DeviceDescriptor_TypeDef` struct to configure the device descriptor.

To obtain a Vendor ID (VID), each manufacturer must obtain a license from USB-IF. In case it is not desired to get their own VID, manufacturers can sub-license a PID from Silicon Labs (for free) and use the Silicon Labs' VID. Instructions can be found here:

<http://www.silabs.com/products/mcu/pages/request-pid.aspx>

3.2 Configuration Descriptor

The configuration descriptors specifies whether the device is self powered (has an external power supply) or bus powered (draws all its power from the bus) as well as the number of interfaces it contains.

The USB specification allows a device to have multiple configuration descriptors. However, the EFM32 USB Stack only supports one configuration descriptor. It is possible to use multiple configuration descriptors, but the device then has to re-enumerate each time the configuration is changed.

The configuration descriptor, as well as the interface and endpoint descriptors are specified in a byte array which is passed to the USB stack. The configuration descriptor is at the beginning of the array, followed by the first interface and its endpoints, followed by the second interface etc.

3.3 Interface Descriptor

An interface in USB context is a set (collection) of endpoints that can be used to communicate with the USB host. A device can have multiple interfaces, which is called a *composite* device. However, the current EFM32 USB Stack does *not* support multiple interfaces. An interface can implement standardized classes (e.g. HID, mass storage device etc.), by specifying the Interface Class (assigned by USB-IF) and defining the required endpoints. Using standardized interfaces reduces the need for custom drivers.

3.4 Endpoint Descriptor

Endpoint descriptors define the address, data direction (IN/OUT) and transfer type (Bulk, Interrupt or Control) of each endpoint.

3.5 String Descriptors

The USB specification allows descriptors to specify human readable strings that identify certain properties of the device, such as the manufacturer's name.

The USB strings are specified in its own descriptor which includes all the strings in unicode format. A descriptor can reference a string by using the index of the string in the string descriptor. A zero value means no string is available. The string descriptor is optional.

The EFM32 USB stack includes convenience macros to allocate the USB strings.

3.6 Endpoint Buffer

The USB controller includes 2 kB of dedicated RAM. This RAM is used to implement FIFO buffers for transfers. When initializing the stack, the application must determine how much RAM to allocate for each endpoint.

To allocate buffers the application must specify a buffering multiplier per endpoint. The buffering multiplier specifies how many times the endpoint size (wMaxPacketSize) the USB stack should allocate for each endpoints FIFO buffer.

A buffering multiplier of 1 (the FIFO is the same size as the endpoint size) is sufficient for Control and Interrupt endpoints. For Bulk endpoints, the buffering multiplier should be 2.

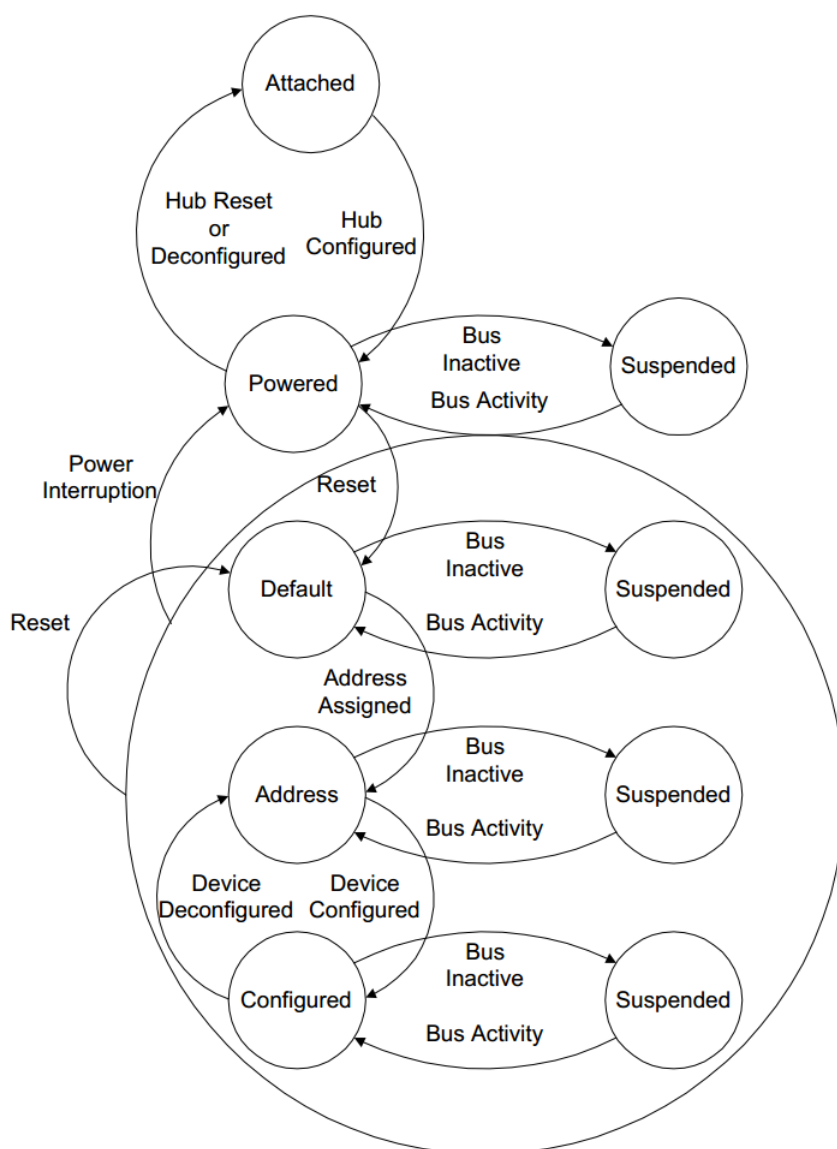
4 State Machine

The USB Stack uses a state machine to keep track of the current state of the USB connection.

The USB Device state machine is shown in Figure 4.1 (p. 7). When the EFM32 is connected to a USB host by a USB cable, the state machine goes through various states until it finally reaches the CONFIGURED state (if the connection was successful). Once in the CONFIGURED state the EFM32 is ready to communicate over USB.

An application-defined callback function is called by the USB stack for every state change. A typical usage of this function is to wait for the CONFIGURED state before starting any communication and use other states (e.g SUSPENDED or DEFAULT) to let application know that communication is no longer possible.

Figure 4.1. EFM32 USB Device State Machine (from [usb_20])



5 USB Device API

This section provides an overview of the most important parts of the EFM32 USB Device API. For more information see the usb Doxygen Documentation.

5.1 Transfer Functions

Apart from the Setup packets, all transfers are initiated by the API functions `USBD_Write()`/`USBD_Read()`. These functions place the transfer in a pending state and return immediately. When the transfer completes (or fails) an application-defined callback function is called.

5.1.1 Buffer Allocation

The read/write functions take a buffer which will be read from or written to. Any buffer passed to these functions should not be allocated on the stack because the functions themselves only prepare the transfer. When the transfer actually occurs, the buffer must still be ready. Therefore the data buffers should be statically allocated.

All buffers passed to `USBD_Write()`/`USBD_Read()` must be WORD (32-bit) aligned. In addition, the USB controller will fill all receive buffers in WORD quantities, so these buffers need to be rounded up to the nearest WORD size. The API provides convenient macros for this. Please see the source example for how to use the macros (`UBUF`, `STATIC_UBUF`, `EFM32_ALIGN`).

For a read (receive) buffer, if it is possible that the host will send more data than expected, round the buffers size up to the next multiple of the max packet size. The USB controller will abort the transfer if the host sends more packets than expected, but it will receive the entire packet that exceeds the expected size.

5.1.2 Callback Functions

When initiating a transfer, the application can specify a callback function which will be called by the USB stack when the transfer has either completed or failed (for instance if the USB cable is unplugged).

The application can use the callback functions to initiate new transfers, take actions based on received data and perform error handling.

5.2 Power Saving and Suspend

When the USB is active and attached to a host the high frequency crystal oscillator (HFXO) must be active and the MCU must be in either EM0 or EM1. When the USB is disconnected or put in the SUSPENDED state, it can be put in a low power mode. In this mode the USB can be clocked by a 32 kHz clock (LFRCO or LFXO) and the MCU can enter EM2.

The USB stack offers a few options on how to handle power saving. These options can be configured at compile time by configuring the `USB_PWRSERVE_MODE` in `usbconfig.h`. This define is a bitmask of zero or more of the following values:

- `USB_PWRSERVE_MODE_ONSUSPEND`
- `USB_PWRSERVE_MODE_ONVBUSOFF`
- `USB_PWRSERVE_MODE_ENTEREM2`

If the `ONSUSPEND` option is set, the USB controller will automatically enter low power mode (clocked by 32 kHz clock) whenever the USB enters suspend mode.

If `ONVBUSOFF` is set, the USB controller will automatically enter low power mode whenever power is lost on VBUS. This requires that the USB regulator is used and that VREG1 is connected to VBUS.

If `ENTEREM2` is set the USB stack will automatically make the MCU enter EM2 whenever USB enters low power mode. If this option is *not* used the application code can instead use the API function `USBD_SafeToEnterEM2()` to determine if it is safe to enter EM2.

The MCU will automatically wake up from EM2 on a USB event (e.g. the MCU cable is plugged in).

See also the Energy-Saving chapter in the USB Device Stack Doxygen Documentation.

5.3 Debug Output

The USB stack can output debug information to a console. To enable debug output, define `DEBUG_USB_API` and `USB_USE_PRINTF` in `usbconfig.h`.

The application should include `retargetio.c` (found in the `kits\common\drivers` directory) and either use `retargetserial.c` to output debug information over UART or use the `setupSWOForPrint()` function from `energyAware Commander` to send the debug information over SWO.

6 Source Code Example

The source code example in this application note is a simple program which configures two endpoints to communicate with a host program on a PC. Source code for the host program is also provided and uses libusb-win32 to access the USB system on a Windows platform.

6.1 EFM32 as USB Device

The EFM32 is configured as a USB device and specifies two endpoints (in addition to EP0)

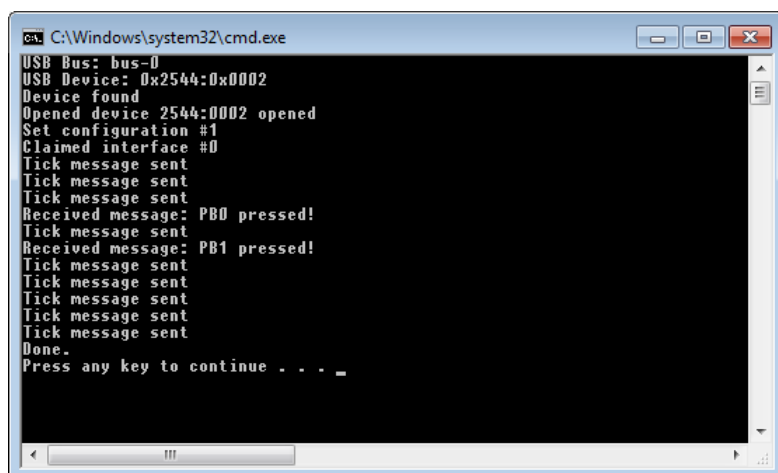
- One interrupt IN endpoint. Used to send messages to the PC when the user presses one of the push buttons on the kit.
- One bulk OUT endpoint. Used to send period messages from the PC to the EFM32. For each message received a counter is incremented and the value is printed on the STK LCD.

6.2 PC as USB Host

The USB host program is a console application for the Windows platform. Project is provided for MSVC as well as a makefile for gcc (tested with MinGW). The host program will enumerate the attached USB devices and attempt to open the device which matches the VID/PID of the device example.

If the device is found, the two pipes are configured and the host program will attempt to send a message on the OUT endpoint every second. If any message is received on the IN endpoint, a message is printed on the screen.

Figure 6.1. USB host program



```
C:\Windows\system32\cmd.exe
USB Bus: bus-0
USB Device: 0x2544:0x0002
Device found
Opened device 2544:0002 opened
Set configuration #1
Claimed interface #0
Tick message sent
Tick message sent
Tick message sent
Received message: PB0 pressed!
Tick message sent
Received message: PB1 pressed!
Tick message sent
Tick message sent
Tick message sent
Tick message sent
Tick message sent
Done.
Press any key to continue . . . _
```

6.3 Install Driver

In order to use a USB device with a non-standard USB class a device driver must be installed first. A driver installer can be found in the folder host\driver. The installer is generated with the program 'inf-wizard.exe' which is part of the libusb-win32 package. This driver will only work with a program linked against the libusb-win32 library.

7 Revision History

7.1 Revision 1.02

2014-05-07

Updated example code to CMSIS 3.20.5

Changed source code license to Silicon Labs license

Added project files for Simplicity IDE

Removed makefiles for Sourcery CodeBench Lite

Changed VID to Silicon Labs

7.2 Revision 1.01

2013-10-31

Added support for EM2 in DK example

7.3 Revision 1.00

2013-08-12

Initial revision.

A Disclaimer and Trademarks

A.1 Disclaimer

Silicon Laboratories intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Laboratories products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Laboratories reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Laboratories shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products must not be used within any Life Support System without the specific written consent of Silicon Laboratories. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Laboratories products are generally not intended for military applications. Silicon Laboratories products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

A.2 Trademark Information

Silicon Laboratories Inc., Silicon Laboratories, Silicon Labs, SiLabs and the Silicon Labs logo, CMEMS®, EFM, EFM32, EFR, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZMac®, EZRadio®, EZRadioPRO®, DSPLL®, ISOmodem®, Precision32®, ProSLIC®, SiPHY®, USBXpress® and others are trademarks or registered trademarks of Silicon Laboratories Inc. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.

B Contact Information

Silicon Laboratories Inc.

400 West Cesar Chavez

Austin, TX 78701

Please visit the Silicon Labs Technical Support web page:

<http://www.silabs.com/support/pages/contacttechnicalsupport.aspx>

and register to submit a technical support request.

Table of Contents

1. Introduction	2
1.1. More Information	2
2. Transfers	3
2.1. Transfer Direction	3
2.2. Endpoints	3
2.3. Transfer Types	3
3. Configuration	5
3.1. Device Descriptor	5
3.2. Configuration Descriptor	5
3.3. Interface Descriptor	6
3.4. Endpoint Descriptor	6
3.5. String Descriptors	6
3.6. Endpoint Buffer	6
4. State Machine	7
5. USB Device API	8
5.1. Transfer Functions	8
5.2. Power Saving and Suspend	8
5.3. Debug Output	9
6. Source Code Example	10
6.1. EFM32 as USB Device	10
6.2. PC as USB Host	10
6.3. Install Driver	10
7. Revision History	11
7.1. Revision 1.02	11
7.2. Revision 1.01	11
7.3. Revision 1.00	11
A. Disclaimer and Trademarks	12
A.1. Disclaimer	12
A.2. Trademark Information	12
B. Contact Information	13
B.1.	13

List of Figures

2.1. Endpoint addresses	3
3.1. USB descriptor hierarchy	5
4.1. EFM32 USB Device State Machine (from [usb_20])	7
6.1. USB host program	10

silabs.com

