SILICON LABORATORIES

AN139

USB FIRMWARE PROGRAMMER'S GUIDE

Relevant Devices

This application note applies to the following devices: C8051F320 and C8051F321.

Introduction

This document serves as a programmer's guide for USB firmware development on Silicon Labs devices. Example firmware is given (starting on page 24). The following discussion is based on the example firmware.

This discussion assumes that the reader is familiar with the USB specification, and is comfortable with terms and abbreviations defined by the USB specification. The following publications are also recommended for USB developers:

1. *USB Complete*, by Jan Axelson

2. *USB Design By Example: A Practical Guide to Building I/O Devices*, by John Hyde

About This Document

Terms with a definition given in the USB specification are displayed with an italicized font: *endpoint*, for example, is a USB specification term. Any instance of the word *endpoint*, when used as defined by the USB specification, is italicized.

Please refer to the USB Specification Terms and Abbreviations chapter for any italicized term; many specification terms have meanings outside of the USB protocol that may be misleading or confusing within the context of this discussion.

Terms with a definition given in the example firmware are displayed in a `code` font. This includes all function, variable, and constant names referenced in the discussion. For a definition of these terms, and a description of the example firmware structure, please see Appendix A beginning on page 18.

What is Firmware Responsible For?

The Silicon Labs USB controller core performs most low-level protocol tasks automatically. A USB interrupt is only generated when data has been received or transmitted successfully, or when a special signaling event (*Reset*, *Resume*, *Stall*, etc.) is detected or generated. Key firmware tasks include the following:

- Decoding and handling incoming *control requests*.
- Preparing data for *IN* transfers.
- Unloading data for *OUT* transfers.
- Managing the USB device state (*Default*, *Addressed*, *Configured*, etc.), and handling requests accordingly.
- Handling *Suspend*, *Reset*, and *Resume* events.

Device Construction

A USB device is defined in terms of *configurations*, *interfaces*, and *endpoints*. Here we discuss each of these and how they are chosen. In this case, it's easiest to start at the bottom (*endpoints*) and work up to the top level device configuration.

Endpoints

Physically, an *endpoint* is a block of FIFO space on a USB device. From the host's perspective, an *endpoint* is either a sink or source of data. *Endpoints* are defined in terms of type (*control*, *interrupt*, *bulk*, or *isochronous*) and direction (*IN* or *OUT*).

On a Silicon Labs device, *endpoints* are implemented in hardware; each *endpoint* has an associated FIFO space and control/status registers.

Endpoint Addressing

The *control endpoint* (Endpoint0) is always bi-directional. All other *endpoints* on Silicon Labs devices are designated with a number and a direction (Endpoint1 IN, Endpoint1 OUT, etc.). Though an *IN*/*OUT endpoint* pair share a number designation, they can be configured to operate independently.

Endpoints are addressed by the host (per the USB protocol) with a bit-mapped byte-wide address, as follows:

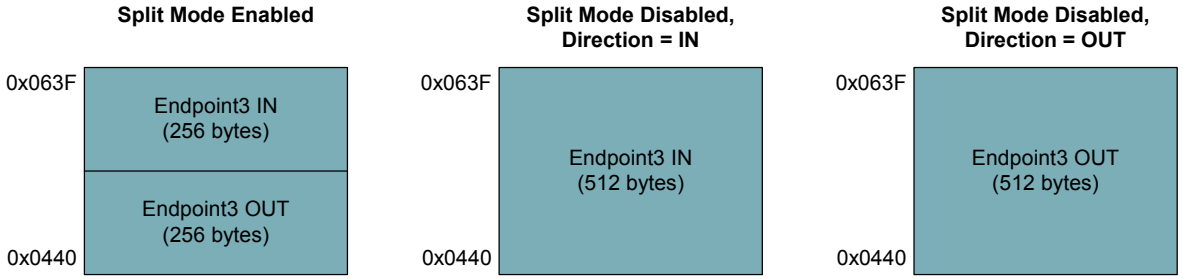
Direction	0	0	0	Endpoint Number			
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

In this address, the direction field is a '0' for *OUT endpoints*, and '1' for *IN endpoints*. Endpoint0 is always bi-directional, and thus has no direction designation.

Endpoint Configuration

The *IN*/*OUT endpoint* pairs on a Silicon Labs device can be configured to operate independently, referred to as Split mode. In Split mode, both the *IN* and *OUT endpoints* for a particular *endpoint* number are enabled, and their shared FIFO space is split. If Split mode is disabled, a single direction (*IN* or *OUT*) must be selected for the *endpoint* pair. An example is shown in Figure 1 for Endpoint3 on a C8051F32x device.

Figure 1. Endpoint FIFO Mode Examples



Once a FIFO mode has been selected, the individual FIFO spaces may also use double packet buffering. When double buffering is enabled, the maximum packet size for a specific *endpoint* is halved, and the FIFO may contain two packets at one time. If an *endpoint* pair is configured for Split mode, a double buffering mode selection must be made for both the *IN* and *OUT endpoints* individually.

Interfaces

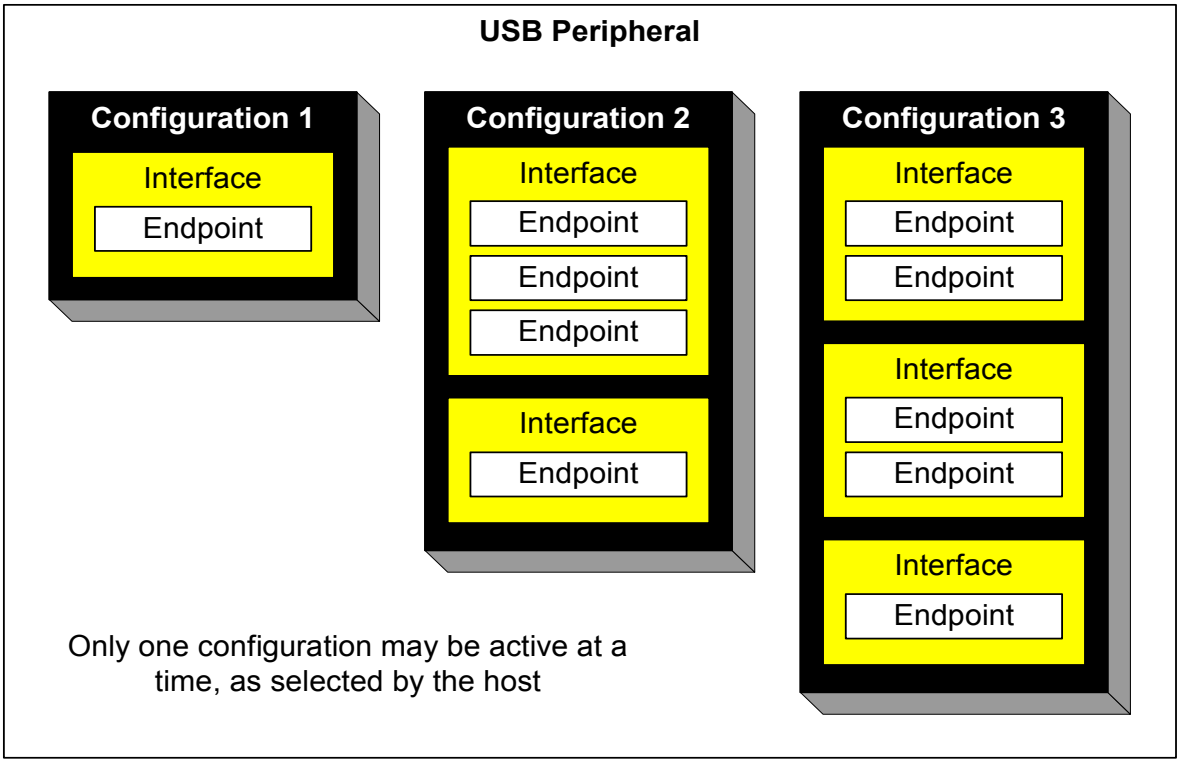
An *interface* is a collection of *endpoints* that work together to implement a particular feature or function. A given USB device may have multiple *interfaces*; however the *interfaces* on a particular device cannot share *endpoints*. *Interfaces* are implemented and managed in firmware, thus the number of *interfaces* on a device is limited only by the number of *endpoints* in the hardware. For most applications, a single *interface* is sufficient.

Configurations

Unlike *interfaces*, only one *configuration* may be active on a device at a time. The *configuration* applies to the device; it includes power features, remote wakeup support status, and a set of *interfaces* and *endpoints*. Typically multiple *configurations* are only necessary when a device supports more than one mode or function. For example, a device that supports firmware upgrade may include one *configuration* for the device's intended function, and another *configuration* to perform the firmware upgrade.

From a programmer's perspective, the complexity of device construction (number of *configurations*, *interfaces*, *alternate interfaces*, etc.) makes writing scalable code extremely difficult. When building a code structure to handle device setup, keep in mind that most applications will only require a single *configuration* and a single *interface*.

Figure 2. Example Configurations



Descriptors

Descriptors are data structures stored in the device that describe the device and its USB capabilities. The host typically requests certain descriptors following a USB *Reset* or when the device is first plugged into the USB network. The host then uses the information in the descriptors to allocate the necessary bus bandwidth and power, and to identify the device when searching for an appropriate driver.

Firmware running on a Silicon Labs device is responsible for storing and transferring USB *descriptors*. Most *descriptors* can be stored in `code` space. *Descriptors* can be easily stored and accessed via an array or a structure. The structure format is acceptable so long as the Compiler does not use padding when locating structure elements. That is, the elements of the structure must be stored in contiguous memory, in the order specified by the structure definition. Use of an array format for *descriptor* storage guarantees that each element will be stored in contiguous memory. The example below shows an array declaration for a *device descriptor* stored in `code` memory:

```
//---------------------------
// Begin Device Descriptor
//---------------------------
BYTE  code  bStdDevDsc[STD_DSC_SIZE]
=
{
   18,                              //
bLength
   0x01,                      // bDe-
scriptorType
    0x00, 0x02,                     //
bcdUSB (lsb first)
   0x00,                      // bDe-
viceClass
   0x00,                      // bDe-
viceSubClass
   0x00,                      // bDe-
viceProtocol
    64,                             //
bMaxPacketSize0
   0xC4, 0x10,                // idV-
endor (lsb first)
    0x00, 0x00,                     //
idProduct (lsb first)
   0x00, 0x00,                // bcd-
Device (lsb first)
    0x00,                           //
iManufacturer
    0x00,                           //
iProduct
   0x00,                      // iSe-
rialNumber
    0x01,                           //
bNumConfigurations
};
// End Device Descriptor
```

Special Signaling Events

Reset, *Suspend*, and *Resume* conditions are considered special signaling events. These events cause an interrupt, if enabled, when detected by hardware; it is the responsibility of firmware to perform any application-related tasks in response.

Reset Events

When a USB *Reset* condition is detected, hardware automatically:

- Resets the *device address* (register FADDR) to 0x00.
- Flushes all *Endpoint* FIFOs.
- Resets Control/Status registers to 0x00 (this includes registers E0CSR, EINCSRL, EINCSRH, EOUTCSRL, EOUTCSRH).
- Resets register INDEX to 0x00.
- Enables all USB interrupts with their interrupt pending flags reset to ‘0’.
- Generates a USB Reset interrupt (if the USB interrupt is enabled).

Firmware must also perform several tasks in response to a USB interrupt. Since the USB device state must be reset to *Default* following a *Reset*, firmware must reset its state machine to the `DEV_DEFAULT` state and assume all *endpoints* (other than Endpoint0) are disabled. Following a USB *Reset*, firmware should:

- Set the device state to DEV_DEFAULT.
- Reset all configuration-related variables (current configuration, number of interfaces, etc.).
- Set the Endpoint0 firmware state to EP_IDLE.
- Set all other *endpoint* firmware states to EP_HALTED.

Suspend and Resume

If Suspend detection and the Suspend interrupt are enabled, hardware generates an interrupt upon detection of *Suspend* signaling on the bus. No other Suspend-related tasks are performed by the hardware; firmware must perform any power saving tasks when this interrupt is generated. Typical power saving options include:

- Using the internal oscillator Suspend Mode. When in this mode, the internal oscillator is stopped until a non-idle USB event is detected. If the internal oscillator is used as the system clock source, code execution will be halted until the oscillator is awakened by a non-idle USB event.
- Disabling the USB Transceiver. *Resume* signaling can still be detected with the Transceiver disabled. Firmware should re-enable the Transceiver immediately following detection of *Resume* signaling.
- VREG Low Power Mode. In low power mode, the on-chip Voltage Regulator current is reduced. This mode should only be used while suspended.

When enabled, a Resume interrupt will be generated when *Resume* signaling is detected.

Endpoint0 Handling (Including Enumeration)

When a USB device is first plugged into a USB network, its USB *device address* is '0', and it is accessible only through its default *control endpoint* (Endpoint0). The host uses Endpoint0 to obtain information about the device via USB *descriptors*. The transfer of these *descriptors*, and the configuration and control of the device by the host, is accomplished through *control transfers* referred to as *Standard Device Requests* (these may also be called *Chapter 9 Requests,* as they are given in Chapter 9 of the USB Specification).

> ***Important Note:*** *References are made throughout this discussion to both firmware states and hardware modes. These states and modes are applicable to each individual endpoint. The following hardware modes are used: Transmit, Receive, and Idle. The following firmware states are used: Idle* (EP_IDLE)*, Error* (EP_ERROR)*, Transmit* (EP_TX)*, and Receive* (EP_RX)*.*

Setup Transactions

Firmware running on a Silicon Labs device is responsible for the management of *control transfers*. Each *control transfer* begins with a *Setup transaction* (a single *packet*) sent from the host to the device. Hardware receives the *Setup packet*, which must contain an 8-byte *command* in the data portion of the *packet*; this *command* field contains the type of request, information necessary for completion of the request, and in some cases all data required for the request. If the data received in the data portion of the *Setup packet* is not 8-bytes (i.e., if the *Setup packet* does not include a complete *command* field), hardware will reject the *Setup packet* without generating an interrupt.

Setup packets are similar to *OUT packets*: hardware sets the OPRDY bit to '1' (to indicate a data packet ready in the FIFO), and generates an interrupt. There is no flag in hardware to indicate that received data in the FIFO is from a *Setup packet* (versus an *OUT packet*); firmware must keep state information that indicates what type of *packet* should be received next. In the example firmware, data in the *OUT* FIFO is assumed to be from a *Setup packet* when the firmware state is EP_IDLE. The firmware state is set to EP_IDLE after any of the following:

1. A device or USB *Reset*.

2. Firmware writes the last outgoing byte to the FIFO (this must be for a *packet* of any size less than the *maximum packet size* for Endpoint0, including a *zero-length packet*).

3. Firmware writes '1' to the SUEND bit after writing a data *packet* to the FIFO.

4. The host sends an *OUT packet* with data less than the *maximum packet size* (including a *zero-length packet*).

5. Hardware sets the SUEND bit to '1' after the host has terminated a *transfer*.

6. Hardware generates an interrupt after sending a *Stall* condition on the bus (this can be a firmware-forced (procedural) or hardware-forced *Stall*).

The nature of the received request will determine the following Endpoint0 Transmit/Receive mode, and thus the direction of the Endpoint0 FIFO. If the request requires the device to send data to the host (data *IN)*, Endpoint0 will enter Transmit mode upon reception of the following *IN token*. If the request requires the host to send data to the device (data *OUT*), Endpoint0 will enter Receive mode upon reception of the following *OUT token*.

Request Handling

Setup packets contain an 8-byte control *command*, which conforms to the following data structure:

```
// Control endpoint command (from
host)
typedef struct EP0_COMMAND
{
   BYTE  bmRequestType;          //
Request type (1 byte)
   BYTE  bRequest;               //
Specific request (1 byte)
   WORD  wValue;                 //
Misc field (2 bytes)
   WORD  wIndex;                 //
Misc index (2 bytes)
   WORD  wLength;                //
Length of the data segment
                                 //
for this request (2 bytes)
} EP0_COMMAND;
```

Upon reception of the *Setup packet*, firmware must decode the 8-byte *command* and respond appropriately. Chapter 9 of the USB specification lists nine *Standard Device Requests* that must be supported by all USB devices and two optional *Standard Device Requests* (see Table 1). The example firmware with this document supports the nine required requests. The *SYNCH_FRAME* and *SET_DESCRIPTOR* requests are not supported; these are handled with a forced *Stall* condition.

Table 1. Standard Device Requests

Request	Direction for Data Stage	Description
GET_CONFIGURATION	IN	The host requests the current device configuration number. Device responds during the data stage with a 1-byte field indicating which configuration was last selected.
GET_STATUS	IN	The host requests the current status of the device, an interface, or an endpoint. The device responds during the data stage with a 2-byte field indicating the requested status.
GET_DESCRIPTOR	IN	The host requests a descriptor (type of descriptor is given in the command field `wValue`). The device responds during the data stage with the requested descriptor data. If the request is for a configuration descriptor, all interface and endpoint descriptors associated with that configuration are returned (up to the amount of data requested by the host).
GET_INTERFACE	IN	The host requests the currently selected alternate for a given interface. The device responds with the alternate number of the currently selected alternate interface.
SET_ADDRESS	no data stage	The host requests the device to change its address to the address contained in the request field `wValue`. No data stage is used for this request. The device updates its address to reflect the new address after the status stage of this request.
SET_CONFIGURATION	no data stage	The host selects a device configuration. Once accepted, the device enters the `DEV_CONFIG` state. No data stage is used for this request.
SET_INTERFACE	no data stage	The host selects an alternate interface for the selected interface number. No data stage is used for this request.
SET_DESCRIPTOR	OUT	The host requests to update or add a descriptor. This request is not required, and is not supported by the Silicon Labs example firmware.
SET_FEATURE	no data stage	The host requests to enable a device feature (remote wakeup) or an endpoint feature (endpoint halt). No data stage is used for this request.
CLEAR_FEATURE	no data stage	The host requests to disable a device feature (remote wakeup) or an endpoint feature (endpoint halt). No data stage is used for this request.
SYNCH_FRAME	IN	This request is used in pattern synchronization between the host and a device supporting isochronous transfers. The device returns a 2-byte frame number during the data stage of this request.

Function Addressing

The host issues a *SET_ADDRESS* request to assign a unique address to the device (called a *device address*). When firmware decodes a *SET_ADDRESS* request, it writes the new address to the FADDR register. Once this address has taken effect, only USB traffic with the same target address will be acknowledged (until the next *USB Reset* or *SET_ADDRESS* request).

Note that the new address will not be effective immediately following the write to FADDR. The *SET_ADDRESS* request includes two *stages* (*Setup* and *Status*); the old address must be used until the entire request is completed. Hardware enables the new address following the *Status stage* of the transfer during which firmware writes to FADDR.

Endpoint0 Receive Mode

When a control request is received that requires the host to transmit data to the device, one or more *OUT* requests will be sent by the host. When an *OUT* packet is successfully received, hardware will set the OPRDY bit to '1' and generate an interrupt. Following this interrupt, firmware should unload the *OUT* packet from the Endpoint0 FIFO and set the SOPRDY bit to '1'.

If the amount of data required for the *transfer* exceeds the *maximum packet size* for Endpoint0, the data will be split into multiple *packets*.

Upon reception of the first *OUT token* for a particular *control transfer*, Endpoint0 is said to be in Receive Mode. In this mode, only *OUT tokens* should be sent by the host to Endpoint0. The SUEND bit is set to '1' if a *SETUP* or *IN token* is received while Endpoint0 is in Receive Mode.

Endpoint0 will remain in Receive mode until the host sends any of the following to the device:

1. *SETUP* or *IN token*.
2. *Packet* less than the *maximum packet size* for Endpoint0.
3. *Zero-length packet*.

Firmware should set the DATAEND bit to '1' when the expected amount of data has been received. Hardware will transmit a *Stall* condition if the host sends an *OUT packet* after the DATAEND bit has been set by firmware. An interrupt will be generated with the STSTL bit set to '1' after the *Stall* is transmitted.

Endpoint0 Transmit Mode

When a control request is received that requires the device to transmit data to the host, one or more *IN transactions* will be initiated by the host. For the first *IN transaction*, firmware should load an *IN packet* into the Endpoint0 FIFO, and set the INPRDY bit to '1'. An interrupt will be generated when an *IN packet* is transmitted successfully. Note that requests received before firmware has loaded a packet into the *endpoint* FIFO will not generate an interrupt. If the requested data exceeds the *maximum packet size* for Endpoint0, as reported to the host, the data should be split into multiple *packets*; each *packet* should be of the *maximum packet size* excluding the last "residual" packet. Firmware should set the DATAEND bit to '1' after loading the last data *packet* for a *transfer* into the Endpoint0 FIFO.

```
   // Check the number of bytes
ready for transmit
   // If less than the maximum
packet size, packet will
   // not be of the maximum size
   if (uNumBytes <= EP0_MAXP)
   {
      uTxBytes = uNumBytes;
      uNumBytes = 0;             //
update byte counter
   }

   // Otherwise, transmit maximum-
length packet
   else
```

```
    {
       uTxBytes = EP0_MAXP;
       uNumBytes -= EP0_MAXP;     //
update byte counter
    }
    // ...Continue to load data into
FIFO
```

Upon reception of the first *IN token* for a particular *control transfer*, Endpoint0 is said to be in Transmit Mode. In this mode, only *IN tokens* should be sent by the host to Endpoint0. The SUEND bit is set to '1' if a *SETUP* or *OUT* token is received while Endpoint0 is in Transmit Mode.

Endpoint0 will remain in Transmit Mode until any of the following occur:

The host sends an Endpoint0 *SETUP* or *OUT* token to the device.

Firmware sends a packet less than the *maximum packet size* for Endpoint0.

Firmware sends a *zero-length packet*.

Firmware should set the DATAEND bit to '1' when performing (2) and (3) above.

Hardware will transmit a *NAK* in response to an *IN token* if there is no *packet* ready in the *IN* FIFO (INPRDY = '0').

Endpoint0 Interrupt Sequencing

Control requests can be one of three flavors: Control Write (data from host to device), Control Read (data from device to host) and Control No Data (no data stage, though some data is transmitted from the host to device in the *Setup packet*). The following figures show the interrupt sequences associated with each type of *control transfer*.

Figure 3. Control Write Sequence

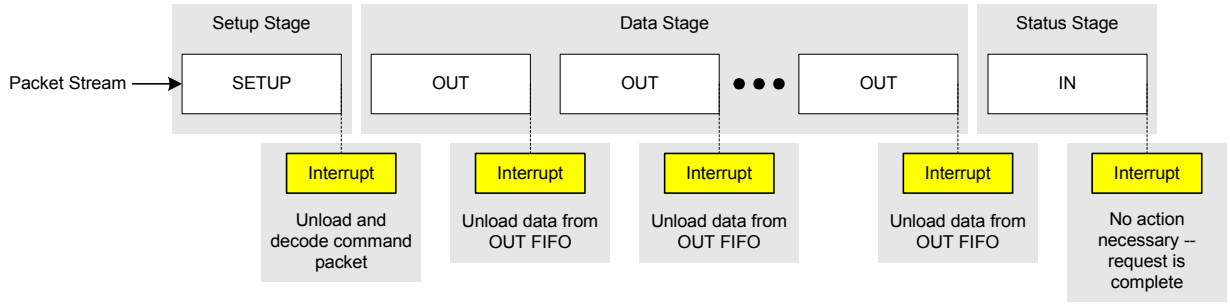


Figure 4. Control Read Sequence

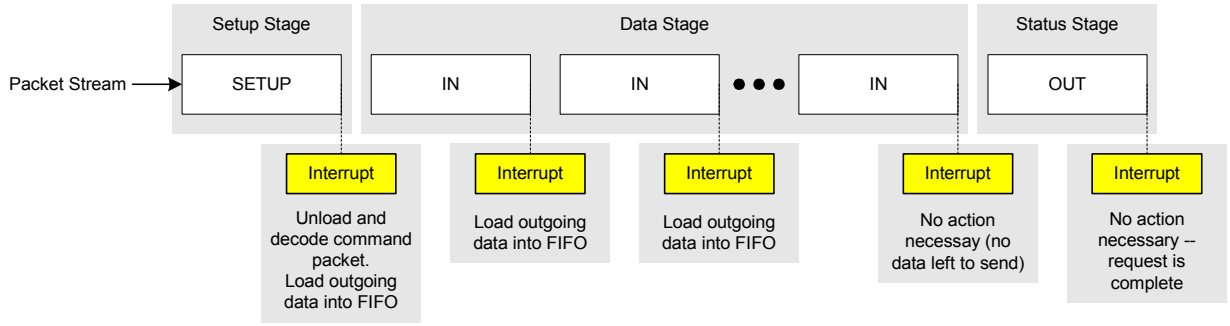
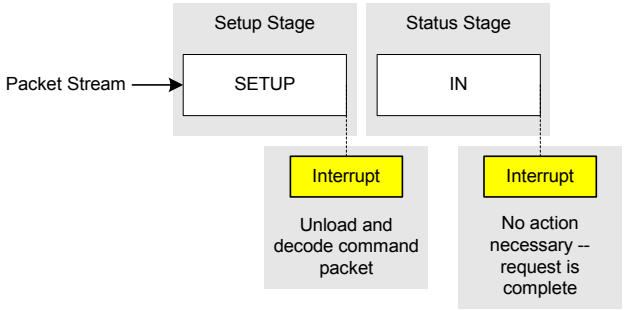


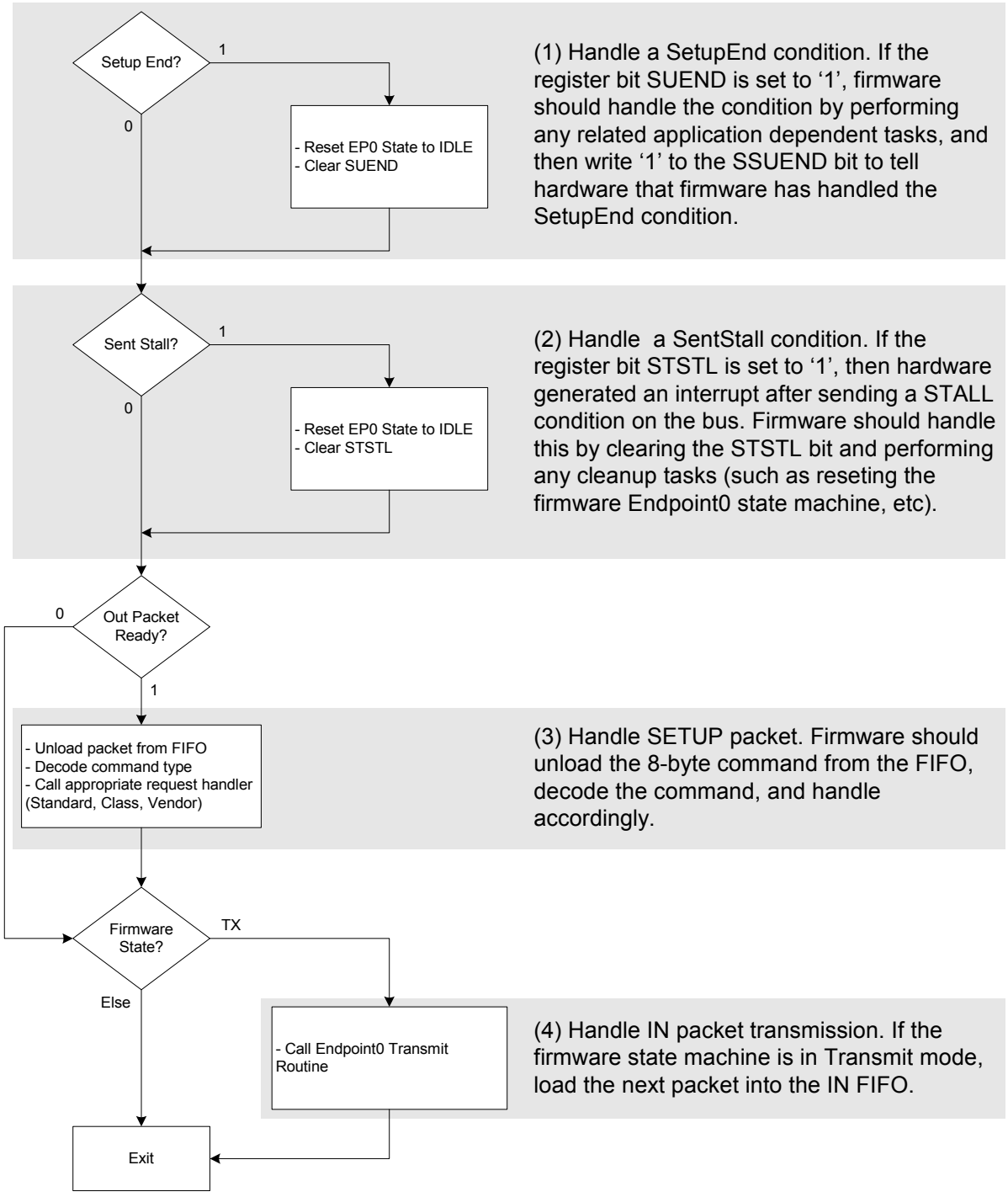
Figure 5. Control No Data Sequence



Endpoint0 Interrupt Handler

The Endpoint0 Interrupt Service Routine (ISR) flowchart is given in Figure 6.

Figure 6. Typical Endpoint0 Handler Summary



Standard Endpoints

Endpoints other than Endpoint0 on a Silicon Labs device are referred to as Standard *endpoints*; each of these operate identically, with the exception that associated FIFO sizes may differ. See the device datasheet for specific *endpoint* FIFO sizes.

Standard *endpoints* support *Interrupt*, *Bulk*, and *Isochronous* transfer modes. Standard *endpoints* are configured and controlled through their own sets of the following control/status registers: *IN* registers EINCSRL and EINCSRH, and *OUT* registers EOUTCSRL and EOUTCSRH. Only one set of *endpoint* control/status registers is mapped into the USB register address space at a time, selected by the contents of the INDEX register.

Bulk, *Interrupt*, and *Isochronous pipes* are configured through communication via Endpoint0. That is, no protocol control information (packet size, transfer frequency, etc.) is sent across one of these *pipes*. All information about a given *pipe* is communicated via standard device requests on Endpoint0. Once the device has entered the `DEV_CONFIG` state (following a *SET_CONFIGURATION* request), the *endpoints* associated with the selected *configuration* may begin communication. Typically the *endpoints* are configured, and set to the firmware state `EP_IDLE`, following a *SET_CONFIGURATION* request.

IN Endpoints

IN endpoints are managed via USB registers EINCSRL and EINCSRH. All *IN endpoints* can be used for *Interrupt*, *Bulk*, or *Isochronous transfers*. Isochronous (ISO) mode is enabled by writing '1' to the ISO bit in register EINCSRH. *Bulk* and *Interrupt transfers* are handled identically by hardware.

An Endpoint1-3 IN interrupt is generated by any of the following conditions:

1. An *IN packet* is successfully transferred to the host.
2. Firmware writes '1' to the FLUSH bit when the target FIFO is not empty.
3. Hardware generates a *Stall* condition.

OUT Endpoints

OUT endpoints are managed via USB registers EOUTCSRL and EOUTCSRH. All *OUT endpoints* can be used for *Interrupt*, *Bulk*, or *Isochronous transfers*. Isochronous (ISO) mode is enabled by writing '1' to the ISO bit in register EOUTCSRH. *Bulk* and *Interrupt* transfers are handled identically by hardware.

An Endpoint1-3 OUT interrupt may be generated by the following:

1. Hardware sets the OPRDY bit to '1'.
2. Hardware generates a *Stall* condition.

Handling Interrupt and Bulk Transfers

This section describes the method for handling *Interrupt* and *Bulk IN* and *OUT transfers*. *Bulk* and *Interrupt transfers* are discussed together because they are handled identically in firmware.

IN Interrupt and Bulk Transfers

Handling an *Interrupt* or *Bulk IN transfer pipe* is fairly simple. The key tasks for the transfer handler are:

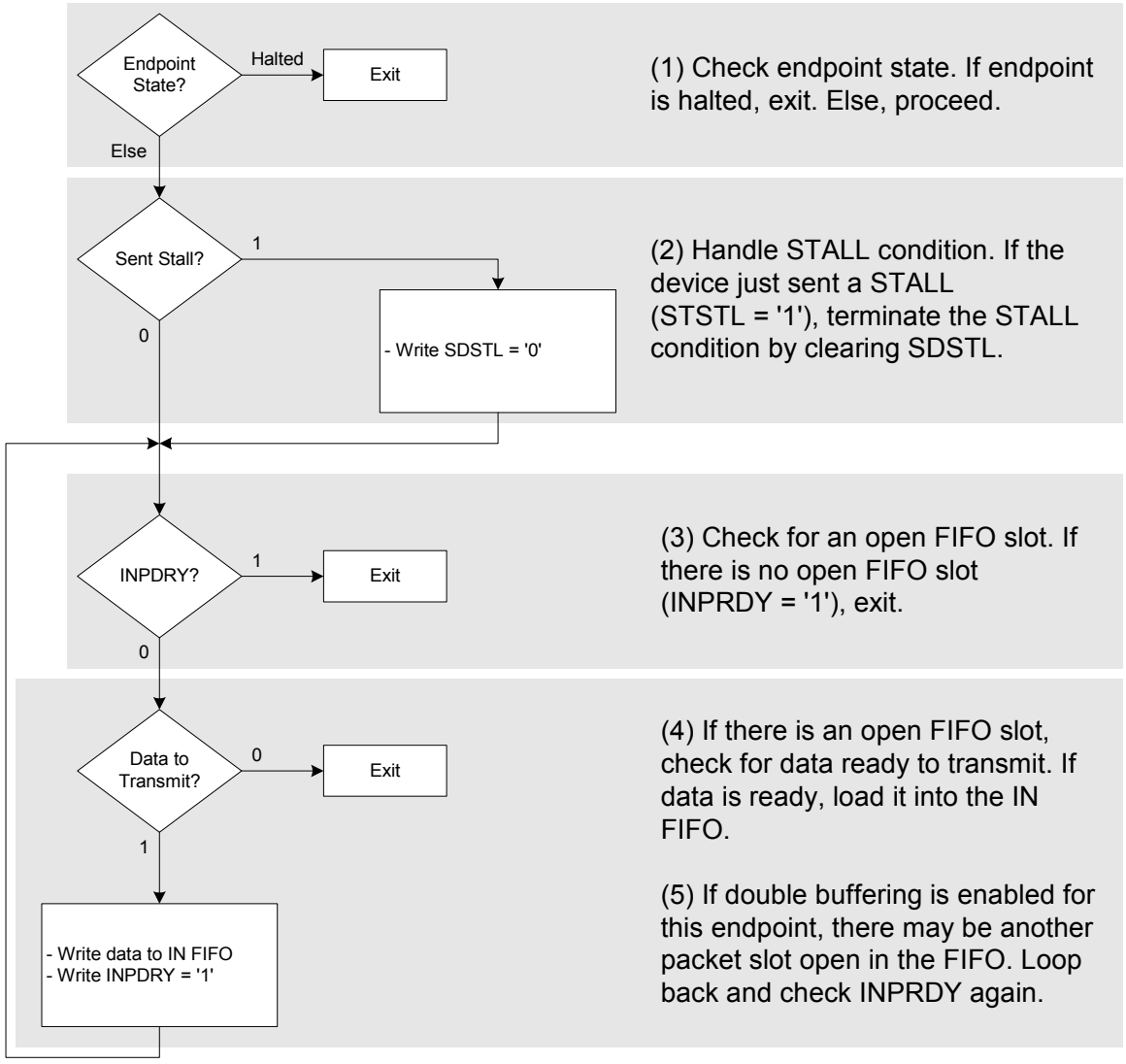
1. Handle *Stall* sent conditions.
2. Manage the packetizing of outgoing data.
3. Load outgoing data into the *IN* FIFO.

The firmware *endpoint* state (structure element `gEp1InStatus.bEpState`) may be one of the following: `EP_IDLE` or `EP_HALTED`. Firmware states `EP_TX` and `EP_ERROR` are not used for this implementation because IN data is always less than the *maximum packet size* and errors are not tracked. Once configured, the *endpoint* should remain in state `EP_IDLE` unless a halted via a *SET_FEATURE* request on Endpoint0. Once in state `EP_HALTED`, a call to the `SetInterface()` routine, a `CLEAR_FEATURE` request received via Endpoint0, or a *Reset* event is required to re-enable the target *endpoint*.

The code segment below performs the transmit portion of the Bulk or Interrupt *IN* handler. Notice that the INPRDY bit should be checked immediately after writing INPRDY = '1'. This is necessary to handle configurations using FIFO double buffering, where the INPRDY bit is immediately cleared to '0' by hardware when a FIFO slot is available.

```
      UREAD_BYTE(EINCSR1, bCsr1);

      // If a FIFO slot is open,
write a new packet to the IN FIFO
      while (!(bCsr1 & rbInINPRDY))
      {
        pEpInStatus->uNumBytes = 8;
              pEpInStatus->pData =
(BYTE*)&IN_PACKET;

         // Write <uNumBytes> bytes
to the <bEp> FIFO
        FIFOWrite(pEpInStatus->bEp,
pEpInStatus->uNumBytes,
                (BYTE*)pEpInStatus-
>pData);

           // Set Packet Ready bit
(INPRDY)
         UWRITE_BYTE(EINCSR1, rbIn-
INPRDY);

          // Check updated endpoint
status
        UREAD_BYTE(EINCSR1, bCsr1);
      }
```

Figure 7. Bulk or Interrupt IN Handler



OUT Interrupt and Bulk Transfers

Handling *Interrupt* or *Bulk OUT transfers* consists of the following tasks:

1. Handle *Stall* sent conditions.

2. Unload incoming data from the *OUT* FIFO.

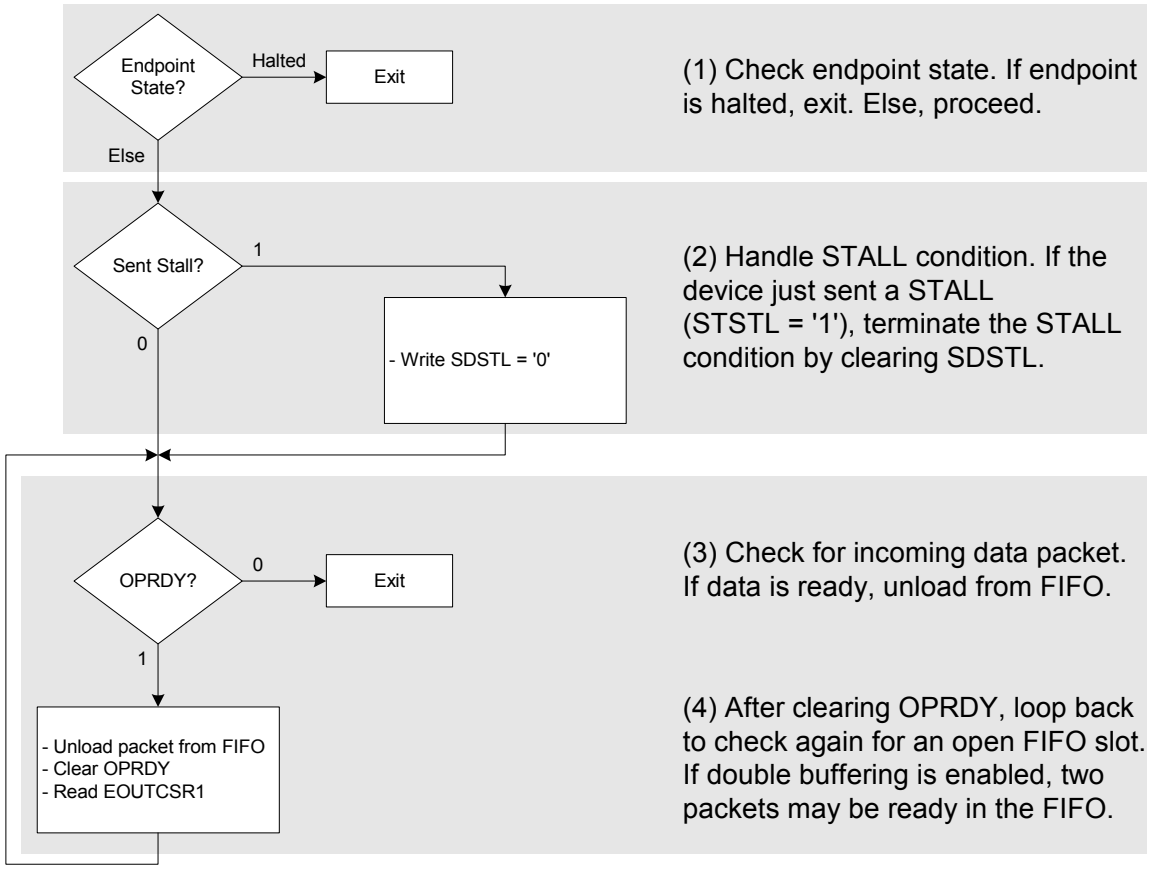
The firmware *endpoint* state (structure element `gEp2OutStatus.bEpState`) may be one of the following: `EP_IDLE` or `EP_HALTED`.

Firmware states `EP_RX` and `EP_ERROR` are not used for this implementation because incoming data is always less than the *maximum packet size* and errors are not tracked. Once configured, the *endpoint* should remain in state `EP_IDLE` unless halted via a *SET_FEATURE* request on Endpoint0. Once in state `EP_HALTED`, a call to the `SetInterface()` routine, a `CLEAR_FEATURE` request received via Endpoint0, or a *Reset* event is required to re-enable the target *endpoint*.

The code segment below performs the receive portion of the Bulk or Interrupt *OUT* handler. Notice that the OPRDY bit should be checked immediately after writing OPRDY = '0'. This is necessary to handle configurations using FIFO double buffering, in which case the OPRDY bit is immediately set to '1' by hardware when a packet is ready in the *OUT* FIFO.

```
UREAD_BYTE(EOUTCSR1, bCsr1);
// Read received packet(s)
//  If double-buffering is enabled,
multiple packets may be ready
while(bCsr1 & rbOutOPRDY)
{
   // Get packet length
   UREAD_BYTE(EOUTCNTL,     bTemp);
// Low byte
   uBytes = (UINT)bTemp & 0x00FF;
   UREAD_BYTE(EOUTCNTH,     bTemp);
// High byte
   uBytes |= (UINT)bTemp << 8;
   if (uBytes == 8)
   {
      // Read FIFO
      FIFORead(pEpOutStatus->bEp,
uBytes, (BYTE*)&OUT_PACKET);
      pEpOutStatus->uNumBytes     =
uBytes;
   }

   // Clear out-packet-ready
   UWRITE_BYTE(EOUTCSR1, 0);

   // Read updated status register
   UREAD_BYTE(EOUTCSR1, bCsr1);
}
```

Figure 8. Bulk or Interrupt OUT Handler



Isochronous Transfers

This section describes the method for handling *Isochronous IN* and *OUT transfers*.

IN Isochronous Transfers

When the ISO bit in register EINCSRH is set to '1', the target *endpoint* operates in Isochronous (ISO) mode. Once an *endpoint* has been configured for ISO *IN* mode, the host will send one *IN token* (data request) per *frame*; the location of data within each *frame* may vary. Because of this, it is recommended that double buffering be enabled for ISO *IN endpoints*.

Hardware will automatically reset INPRDY to '0' when a packet slot is open in the *endpoint* FIFO. Note that if double buffering is enabled for the target *endpoint*, it is possible for firmware to load two *packets* into the *IN* FIFO at a time. In this case, hardware will reset INPRDY to '0' immediately after firmware loads the first *packet* into the FIFO and sets INPRDY to '1'. An interrupt will not be generated in this case; an interrupt will only be generated when a data *packet* is transmitted. If there is not a data *packet* ready in the *endpoint* FIFO when hardware receives an *IN token* from the host, hardware will transmit a *zero-length data packet* and set the UNDRUN bit to '1' indicating a data underrun.

The ISO Update feature can be useful in starting a double buffered ISO *IN endpoint*. If the host has already set up the ISO *IN* pipe (has begun transmitting *IN tokens*) when firmware writes the first data *packet* to the *endpoint* FIFO, the next *IN token* may arrive and the first data packet sent before firmware has written the second (double buffered) data *packet* to the FIFO. The ISO Update feature ensures that any data *packet* written to the *endpoint* FIFO will not be transmitted during the current frame; the *packet* will only be sent after a SOF signal has been received.

OUT Isochronous Transfers

When the ISO bit in register EOUTCSRH is set to '1', the target *endpoint* operates in Isochronous (ISO) mode. Once an *endpoint* has been configured for ISO *OUT* mode, the host will send exactly one data *packet* per *frame*; the location of the data *packet* within each *frame* may vary, however. Because of this, it is recommended that double buffering be enabled for ISO *OUT endpoints*.

Each time a data *packet* is received, hardware will load the received data *packet* into the *endpoint* FIFO, set the OPRDY bit in register EOUTCSRL to '1', and generate an interrupt, if enabled. Firmware would typically use this interrupt to unload the data packet from the *endpoint* FIFO and reset the OPRDY bit to '0'.

If a data *packet* is received when there is no room in the *endpoint* FIFO, an interrupt will be generated and the OVRUN bit set to '1'. If hardware receives an ISO data *packet* with a CRC error, the data *packet* will be loaded into the *endpoint* FIFO, OPRDY will be set to '1', an interrupt (if enabled) will be generated, and the DATAERR bit will be set to '1'. Software should check the DATAERR bit each time a data *packet* is unloaded from an ISO *OUT endpoint* FIFO.

Appendix A—Program Structure

This section describes the example firmware included with the Programmer's Guide.

Notation

Most variable names follow a form of Hungarian Notation. The following table includes a description of each type of variable or constant name prefix:

Table 2. Variable and Constant Prefixes

Prefix	Description
b	BYTE (unsigned char)
u	UINT (unsigned int)
p	Pointer
w	WORD (two bytes, organized as a union {unsigned int i; unsigned char c[2];})
g	Global variable
rb	A bit mask for bits in a Common, Interrupt, or Endpoint0 USB register. These are defined in file "usb_regs.h" For example, rbOPRDY refers to bit OPRDY in Endpoint0 register E0CSR.
rbIn	A bit mask for bits in an IN Status USB register (EINCSR1 or EINCSR2). These are defined in file "usb_regs.h" For example, rbInINPRDY refers to bit INPRDY in register EINCSR1.
rbOut	A bit mask for bits in an OUT Status USB register (EOUTCSR1 or EOUTCSR2). These are defined in file "usb_regs.h". For example, rbOutOPRDY refers to bit OPRDY in register EOUTCSR1.
bm	Bit-mapped BYTE. This is only used in the Endpoint0 Command structure (EP0_COMMAND).
std_	Array offset for a standard device descriptor field. For example, StdDescriptor[std_bLength] is used to access the bLength field of standard device descriptor array StdDescriptor[].
cfg_	Array offset for a configuration descriptor field. For example, ConfigDescriptor[cfg_bLength] is used to access the bLength field of configuration descriptor array ConfigDescriptor[].
if_	Array offset for an interface descriptor field. For example, IfDescriptor[if_bLength] is used to access the bLength field of interface descriptor array IfDescriptor[].
ep_	Array offset for an endpoint descriptor field. For example, EpDescriptor[ep_bLength] is used to access the bLength field of endpoint descriptor array EpDescriptor[].

Header Files

Several header (*.h) files are used with the firmware. Each is described here.

usb_main.h

Standard and configuration constants are defined in file usb_main.h, along with device and endpoint states. Function prototypes are also listed in this file. Constants referenced in this document are defined below:

Table 3. Constant Descriptions in usb_main.h (not all inclusive)

Constant	Description
DEV_DEFAULT	Device State Default - The device state variable (gDeviceStatus.bDevState) is set to DEV_DEFAULT following a device or USB reset event. In this state, USB communication is only allowed via Endpoint0, and some Standard Device Requests are disallowed.
DEV_ADDRESS	Device State Addressed - The device state variable is set to DEV_ADDRESS when a USB address (other than 0x00) is received via a SET_ADDRESS request on Endpoint0. In this state, USB communication is only allowed via Endpoint0, and some Standard Device Requests are disallowed.
DEV_CONFIG	Device State Configured - The device state variable is set to DEV_CONFIG following a SET_CONFIGURATION request on Endpoint0. In this state, USB communication via configured endpoints is allowed (note that a SET_INTERFACE request is also required to configure standard endpoints; however, the device state does not change following a SET_INTERFACE request). The device state remains DEV_CONFIG until a device reset event, a USB reset event, or a SET_CONFIGURATION request for configuration '0'.
EP_IDLE	Endpoint State Idle - The firmware endpoint state is set to EP_IDLE when the endpoint is in between transfers.
EP_TX	Endpoint State Transmit - The firmware endpoint state is set to EP_TX when the endpoint is in the process of transmitting a block of data (may last through several packets of data).
EP_RX	Endpoint State Receive - The firmware endpoint state is set to EP_RX when the endpoint is in the process of receiving a block of data (multiple packets). Reception of a single data packet will not cause the endpoint to enter the EP_RX state.
EP_ERROR	Endpoint State Error - The firmware endpoint state is set to EP_ERROR when a protocol error has been detected (a bad Standard Device Request, for example). The endpoint handler uses this information to respond to the error (typically by transmitting a Stall condition); the endpoint state is set back to EP_IDLE when appropriate.
EP_HALTED	Endpoint State Halted - The firmware endpoint state is set to EP_HALTED when an endpoint (other than Endpoint0) is halted via a SET_FEATURE request on Endpoint0. Once halted, the endpoint will remain in this state until enabled via a CLEAR_FEATURE request, a SET_INTERFACE request, or a USB or device reset event.

usb_config.h

File usb_config.h contains constants defined based on the USB device configuration (number of configurations, number of interfaces, remote wakeup support, etc.). These constants can be modified to allow the firmware framework to be used with different device configurations.

usb_desc.h

File usb_desc.h contains array offset definitions for use with descriptor arrays. For example, the `bMaxPower` field of the `gDescriptorMap.bCfg1` configuration descriptor array is accessed as follows:

```
x = gDescriptorMap.bCfg1[cfg_bMaxPower]
```

Device, interface, and endpoint descriptor fields are accessed in a similar manner. See Table 2 for array constant prefix descriptions.

usb_request.h

Codes and bit masks relating to *Standard Device Request* handling are defined in file usb_request.h.

usb_structs.h

All data structures used in the example firmware are defined in file usb_structs.h. The following table describes these structures.

Table 4. Structure Descriptions

Structure	Instances	Description
`DEVICE_STATUS`	`gDeviceStatus`	Information about the state of the USB device, including the current configuration, address of the current configuration descriptor, number of interfaces for this configuration, self/bus powered and remote wakeup status.

`EP_STATUS`	`gEp0Status`
	`gEp1InStatus`
	`gEp2OutStatus`
		This structure is used for IN endpoints, OUT endpoints, and Endpoint0. Information here includes the endpoint number (or address), endpoint state, maximum packet size, number of bytes to load (IN) or unload (OUT) to/from the endpoint FIFO, and a pointer to outgoing data or a target location for incoming data.

`EP0_COMMAND`	`gEp0Command`	An 8-byte Command (or request) from an Endpoint0 Setup packet, as defined in Chapter 9 of the USB specification.
`DESCRIPTORS`	`gDescriptorMap`	Includes two arrays: one for the Device descriptor, and one for Configuration 1 (includes the configuration, interface, and endpoint descriptors for Configuration 1).
`IF_STATUS`	`Member of DEVICE_STATUS`	Information about a particular interface within a configuration: Interface number, number of alternates, and the current alternate.

usb_regs.h

This file includes the regsiter addresses used to access USB controller registers. Bit masks for these registers are also included; these bit masks use a prefix naming scheme described in Table 2. The Macros used to access USB controller registers are also included in this file; see Section “ Accessing USB Controller Registers” on page 23 for a description of these Macros.

Firmware Architecture

Key pieces of a USB firmware package are the following:

- Initialization code (clock, Ports, USB core, etc.)
- Top level USB Interrupt Service Routine (ISR) and Reset Handler
- Endpoint0 Handler
- Standard Device Request Handlers
- Standard Device Request Helper Routines
- Additional *Endpoint* Handlers

The example firmware included with this document is organized as follows:

main.c

All initialization routines are located in file main.c, along with the ADC0 ISR used to monitor the temperature and potentiometer for the Windows application.

usb_desc.c

Descriptor declarations are given in file usb_desc.c.

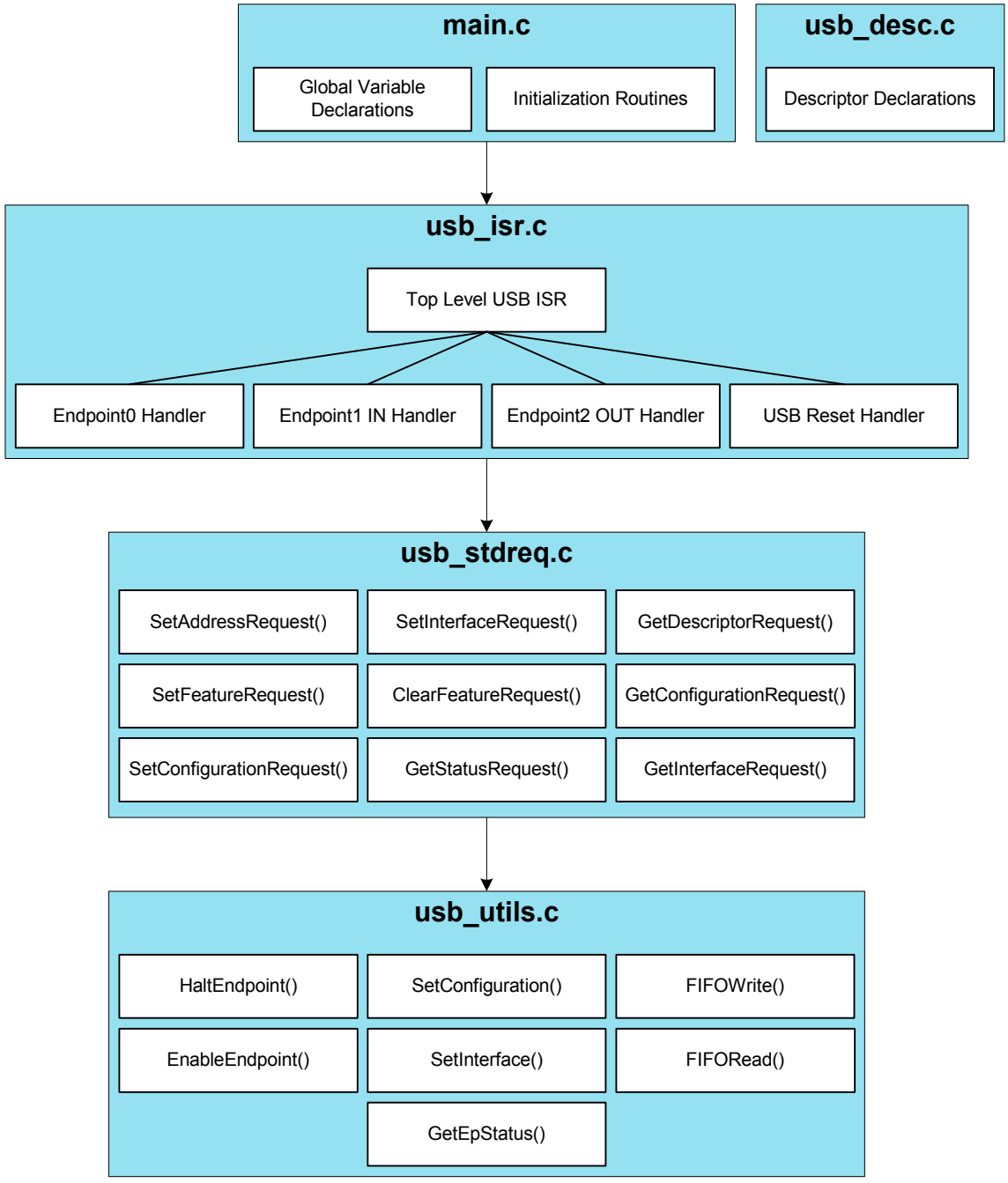
usb_isr.c

All USB-related interrupt handlers are included in file usb_isr.c: Top Level USB ISR, Endpoint0 Handler, Bulk or Interrupt *Endpoint IN* Handler, and Bulk or Interrupt *Endpoint OUT* Handler. Since hardware supports a single USB interrupt, the Top Level ISR calls into the *endpoint* handlers or the Reset handler (in usb_reset.c). The Endpoint0 Handler decodes incoming requests and handles requests by calling the appropriate standard device request routines (in usb_stdreq.c).

usb_stdreq.c

Handler routines for all standard device requests are included in file usb_stdreq.c. These routines are called by the Endpoint0 Handler and modify fields of the `gEp0Status` structure to indicate the result of these requests. For Read requests (data stage from device to host), the `bEpState` field will be set to `EP_TX`, the `pData` field set to the data source location, and the `uNumBytes` field set to the number of bytes to transfer. Write requests (data stage from host to device) are not supported. For "no data" transfer requests, the `bEpState` field is left as `EP_IDLE`. If an error occurs in the request (i.e., an illegal request), the `bEpState` field is set to `EP_ERROR`. The Endpoint0 Handler uses these modified fields of the `gEp0Status` structure to determine how the request will be completed.

Figure 9. Program Architecture



Accessing USB Controller Registers

USB controller registers are accessed via two Special Function Registers (SFRs): USB0ADR and USB0DAT. This method is described in detail in the USB section of the device datasheet. The example firmware uses the following Macros to access all USB registers, defined in file usb_regs.h:

```
// Register read/write macros
#define UREAD_BYTE(addr,target)
USB0ADR = (0x80 | addr);
   while(USB0ADR & 0x80); target =
USB0DAT

#define UWRITE_BYTE(addr,data)
USB0ADR = addr; USB0DAT = data;
   while(USB0ADR & 0x80)
```

Accessing Endpoint FIFOs

Endpoint FIFOs are accessed using the above Macros targeting the FIFOn registers (where n is the *endpoint* number). A write to the FIFOn register loads one byte into the *endpoint* FIFO; a read of the FIFOn register unloads one byte from the *endpoint* FIFO. Two FIFO access routines are given in file "usb_utils.c": `FIFORead()` and `FIFOWrite()`. These routines accept an *endpoint* number, number of bytes, and target/source address for block data transfers. The `FIFORead()` routine utilizes the Auto Read bit (AUTORD) for faster block reads, as follows:

```
USB0ADR = (TargetReg & 0x3F);     //
Set address (mask out bits7-6)
USB0ADR |= 0xC0;                  //
Set auto-read and initiate
                                  //
first read

// Unload <NumBytes - 1> from the
selected FIFO
for( i=0; i<(uNumBytes-1); i++)
{
   while(USB0ADR & 0x80);         //
Wait for BUSY->'0' (data ready)
   pData[i] = USB0DAT;            //
Read data byte
}

// Disable auto read and copy last
byte
USB0ADR = (TargetReg & 0x3F);     //
Set address (mask out bits7-6)
while(USB0ADR & 0x80);            //
Wait for BUSY->'0' (data ready)
pData[i] = USB0DAT;               //
Copy final data byte
```

Appendix B—Firmware Listing

A source code listing for the entire USB firmware example is given below; this includes both ‘C’ code and required header (*.h) files used in the project.

main.c

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:     main.c
   Author:   JS
   Created:  JAN 03
   Modified: JUL 03 -- BW (changed 'while(!(CLKMUL & 0x20));')


   Target Device: C8051F320

   Source file for USB firmware. Includes main routine and
   all hardware initialization routines.

   This is the main routine file for the USBProgramGuide.wsp project,
   which includes the following source files:

   main.c
   usb_desc.c
   usb_isr.c
   usb_stdreq.c
   usb_utils.c

   This firmware is intended to work with the Cygnal USB Test Application,
   implementing two Interrupt pipes with 8-byte transfers. The endpoints
   used are as follows:

   Endpoint1 IN - Interrupt IN
   Endpoint2 OUT - Interrupt OUT

   The transmit buffer (used with IN packets) is the 8-byte array
   <IN_PACKET>. This buffer is updated in the main loop. The receive
   buffer (used with OUT packets) is the 8-byte array <OUT_PACKET>. The
   information in received packets is used to update Port I/O in the
   main loop.

****************************************/

#include <c8051f320.h>                    // SFR declarations
#include "usb_regs.h"
#include "usb_structs.h"
#include "usb_main.h"
#include "usb_desc.h"

//-----------------------------------------------------------------------
// Global Constants
//-----------------------------------------------------------------------

sbit LED1 = P2^2;                         // LED='1' means ON
```

```
sbit LED2 = P2^3;
sbit BTN1 = P2^0;                      // Target board button 1
sbit BTN2 = P2^1;                      // Target board button 2

//-----------------------------------------------------------------------------
// Local Function Prototypes
//-----------------------------------------------------------------------------

void USB0_Init (void);
void USB0_Enable (void);
void PORT_Init (void);
void SYSCLK_Init (void);
void Timer2_Init (void);
void ADC_Init (void);

//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------

DEVICE_STATUS gDeviceStatus;
EP_STATUS gEp0Status;
EP_STATUS gEp1InStatus;
EP_STATUS gEp2OutStatus;
EP0_COMMAND gEp0Command;

BYTE OUT_PACKET[8]   = {0,0,0,0,0,0,0,0}; // Last packet received from host
BYTE IN_PACKET[8] = {0,0,0,0,0,0,0,0};    // Next packet to sent to host

//-----------------------------------------------------------------------------
// Main Routine
//-----------------------------------------------------------------------------

void main (void)
{
   BYTE LastBTN1, LastBTN2 = 0;

   PCA0MD &= ~0x40;                    // Disable Watchdog timer

   PORT_Init ();                       // Initialize Crossbar and GPIO
   SYSCLK_Init ();                     // Initialize oscillator
   Timer2_Init ();                     // Initialize timer2
   ADC_Init ();                        // Initialize ADC
   USB0_Init ();                       // Initialize USB0

   EA = 1;                             // Enable global interrupts

   USB0_Enable ();                     // Enable USB0

   // Main loop updates the IN_PACKET with Port I/O and analog
   // measurement status, and updates Port I/O with information
   // from the OUT_PACKET.
   while (1)
   {
      LED1 = OUT_PACKET[0];            // Update status of LED #1
      LED2 = OUT_PACKET[1];            // Update status of LED #2
      P1 = (OUT_PACKET[2] & 0x0F);     // Update Port1 outputs

      // If button 1 is pressed and its previous state was open,
      // toggle the button 1 flag.
```

```
      if ((BTN1 == 0) && (LastBTN1 == 1))
      {
         IN_PACKET[0] = ~IN_PACKET[0];
      }
      LastBTN1 = BTN1;                    // Store the current button state

      // If button 2 is pressed and its previous state was open,
      // toggle the button 2 flag.
      if ((BTN2 == 0) && (LastBTN2 == 1))
      {
         IN_PACKET[1] = ~IN_PACKET[1];
      }
      LastBTN2 = BTN2;                    // Store the current button state

      IN_PACKET[2] = (P0 & 0x0F);         // Update Port0 [0-3] indicators

      // ADC data updated in ADC0 ISR
   }
}

//-----------------------------------------------------------------------
// Interrupt Service Routines (non-usb)
//-----------------------------------------------------------------------

//---------------------------
// ADC0_ISR
//---------------------------
// - Record conversion data
// - Update MUX for next conversion
//
void ADC0_ISR (void) interrupt 10
{
   // Handle Temperature Sensor measurement
   if (AMX0P == AMX_TEMP_SENSE)
   {
      IN_PACKET[4] = ADC0H;               // Update IN packet
      AMX0P = AMX_P1_7;                   // Update AMUX
   }

   // Handle Potentiometer measurement
   else
   {
      IN_PACKET[3] = ADC0H;               // Update IN packet
      AMX0P = AMX_TEMP_SENSE;             // Update AMUX
   }

   AD0INT = 0;

}

//-----------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------

//---------------------------
// SYSCLK_Init
//---------------------------
// SYSLCK Initialization
// - Initialize the system clock and USB clock
```

```
//
void SYSCLK_Init (void)
{
   unsigned char delay = 100;

   OSCICN |= 0x03;                     // Configure internal oscillator for
                                       // its maximum frequency

   CLKMUL = 0x00;                      // Select internal oscillator as
                                       // input to clock multiplier

   CLKMUL |= 0x80;                     // Enable clock multiplier
   while (delay--);                    // Delay for >5us
   CLKMUL |= 0xC0;                     // Initialize the clock multiplier

   while(!(CLKMUL & 0x20));            // Wait for multiplier to lock

   CLKSEL |= USB_4X_CLOCK;             // Select USB clock
   CLKSEL |= SYS_INT_OSC;              // Select system clock
}


//----------------------------
// USB0_Init
//----------------------------
// USB Initialization
// - Initialize USB0
// - Enable USB0 interrupts
// - Enable USB0 transceiver
// - USB0 left disabled
//
void USB0_Init (void)
{
   UWRITE_BYTE(POWER, 0x08);           // Asynch. reset

   UWRITE_BYTE(IN1IE, 0x0F);           // Enable Endpoint0 Interrupt
   UWRITE_BYTE(OUT1IE, 0x0F);
   UWRITE_BYTE(CMIE, 0x04);            // Enable Reset interrupt

   USB0XCN = 0xC0;                     // Enable transceiver
   USB0XCN |= FULL_SPEED;              // Select device speed

   UWRITE_BYTE(CLKREC, 0x80);          // Enable clock recovery,
                                       // single-step mode disabled

   EIE1 |= 0x02;                       // Enable USB0 Interrupts
}

//----------------------------
// USB0_Enable
//----------------------------
// USB Enable
// - Enable USB0
//
void USB0_Enable (void)
{

   UWRITE_BYTE(POWER, 0x00);           // Enable USB0 by clearing the
                                       // USB Inhibit bit
```

```
                                       // Suspend mode disabled
}

//----------------------------
// ADC_Init
//----------------------------
// ADC initialization
// - Single-ended mode
// - Conversions on Timer2 overflows
// - Left-justified data
// - Low power tracking mode
// - Interrupts enabled
//
void ADC_Init (void)
{
   REF0CN = 0x0E;                      // Temperature sensor enabled,
                                       // internal bias enabled,
                                       // internal voltage reference
                                       // enabled

   AMX0P = AMX_TEMP_SENSE;             // Temp sensor as positive input
   AMX0N = AMX_GND;                    // GND as negative input
                                       // (single-ended)

   ADC0CF = (SYSCLK/2000000) << 3;     // SAR clock set to 2MHz
   ADC0CF |= 0x04;                     // Left-justified data

   ADC0CN = 0xC2;                      // ADC0 enabled, low power
                                       // tracking mode,
                                       // Timer2 conversions

   EIE1 |= 0x08;                       // Enable conversion complete
                                       // interrupt
}

//----------------------------
// Timer2_Init
//----------------------------
// Timer initialization
// - Timer2 configured to overflow at the frequency defined
// by <SAMPLERATE> for ADC conversions
//
void Timer2_Init ()
{
   TMR2CN  = 0x00;                     // Stop Timer2; Clear TF2;
   CKCON  |= 0x10;                     // SYSCLK as Timer2 clock source
   TMR2RL  = -(SYSCLK/SAMPLERATE);     // Timer2 overflows at <SAMPLERATE>
   TMR2    = 0xFFFF;                   // set to reload immediately

   TR2     = 1;                        // start Timer2
}

//----------------------------
// PORT_Init
//----------------------------
// Port Initialization
// - Configure the Crossbar and GPIO ports.
//
void PORT_Init (void)
```

```
{
   P1MDIN   = 0x7F;                     // P0.7: analog input
   P0MDOUT  = 0x0F;                     // P0.0-P0.3: push-pull
   P1MDOUT  = 0x0F;                     // P1.0-P1.3: push-pull
   P2MDOUT  = 0x0C;                     // P2.3-P2.2: push-pull
   P1SKIP   = 0x80;                     // P1.7 skipped by Crossbar
   XBR0     = 0x00;                     //
   XBR1     = 0x40;                     // Enable Crossbar
}
```

usb_desc.c

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_desc.c
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Source file for USB firmware. Includes descriptor data structures
   (device, configuration, interface, endpoint).

   Functions:
   None

*****************************************/

#include "usb_main.h"
#include "usb_structs.h"
#include "usb_regs.h"
#include "usb_request.h"
#include "usb_desc.h"

//----------------------------
// Descriptor Declarations
//----------------------------
// All descriptors are contained in the global structure <gDescriptorMap>.
// This structure contains BYTE arrays for the standard device descriptor
// and all configurations. The lengths of the configuration arrays are
// defined by the number of interface and endpoint descriptors required
// for the particular configuration (these constants are named
// CFG1_IF_DSC and CFG1_EP_DSC for configuration1).
//
// The entire gDescriptorMap structure is initialized below in
// codespace.

DESCRIPTORS code gDescriptorMap = {

//----------------------------
// Begin Standard Device Descriptor (structure element stddevdsc)
//----------------------------
   18,                         // bLength
   0x01,                       // bDescriptorType
   0x00, 0x02,                 // bcdUSB (lsb first)
   0x00,                       // bDeviceClass
   0x00,                       // bDeviceSubClass
   0x00,                       // bDeviceProtocol
   64,                         // bMaxPacketSize0
   0xC4, 0x10,                 // idVendor (lsb first)
   0x00, 0x00,                 // idProduct (lsb first)
   0x00, 0x00,                 // bcdDevice (lsb first)
   0x00,                       // iManufacturer
   0x00,                       // iProduct
   0x00,                       // iSerialNumber
   0x01,                       // bNumConfigurations

//----------------------------
```

```
// Begin Configuration 1 (structure element cfg1)
//---------------------------

   // Begin Descriptor: Configuration 1
   0x09,                       // Length
   0x02,                       // Type
   0x20, 0x00,                 // TotalLength (lsb first)
   0x01,                       // NumInterfaces
   0x01,                       // bConfigurationValue
   0x00,                       // iConfiguration
   0x80,                       // bmAttributes (no remote wakeup)
   0x0F,                       // MaxPower (*2mA)

   // Begin Descriptor: Interface0, Alternate0
   0x09,                       // bLength
   0x04,                       // bDescriptorType
   0x00,                       // bInterfaceNumber
   0x00,                       // bAlternateSetting
   0x02,                       // bNumEndpoints
   0x00,                       // bInterfaceClass
   0x00,                       // bInterfaceSubClass
   0x00,                       // bInterfaceProcotol
   0x00,                       // iInterface

   // Begin Descriptor: Endpoint1, Interface0, Alternate0
   0x07,                       // bLength
   0x05,                       // bDescriptorType
   0x81,                       // bEndpointAddress (ep1, IN)
   0x03,                       // bmAttributes (Interrupt)
   0x40, 0x00,                 // wMaxPacketSize (lsb first)
   0x05,                       // bInterval

   // Begin Descriptor: Endpoint2, Interface0, Alternate0
   0x07,                       // bLength
   0x05,                       // bDescriptorType
   0x02,                       // bEndpointAddress (ep2, OUT)
   0x03,                       // bmAttributes (Interrupt)
   0x40, 0x00,                 // wMaxPacketSize (lsb first)
   0x05,                       // bInterval

//---------------------------
// End Configuration 1
//---------------------------

};
```

usb_isr.c

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.
   File:    usb_isr.c
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Source file for USB firmware. Includes the following routines:

   - USB_ISR(): Top-level USB interrupt handler. All USB handler
      routines are called from here.
   - Endpoint0(): Endpoint0 interrupt handler.
   - BulkOrInterruptOut(): Bulk or Interrupt OUT interrupt
      handler.
   - BulkOrInterruptIn(): Bulk or Interrupt IN interrupt
      handler.
   - USBReset (): USB Reset event handler.

****************************************/

#include "c8051F320.h"
#include "usb_regs.h"
#include "usb_structs.h"
#include "usb_main.h"
#include "usb_desc.h"
#include "usb_config.h"
#include "usb_request.h"

extern DEVICE_STATUS gDeviceStatus;
extern EP0_COMMAND gEp0Command;
extern EP_STATUS gEp0Status;
extern EP_STATUS gEp1InStatus;
extern EP_STATUS gEp2OutStatus;
extern BYTE OUT_PACKET[];
extern BYTE IN_PACKET[];

//----------------------------
//  USB ISR
//----------------------------
//
// This is the top level USB ISR. All endpoint interrupt/request
// handlers are called from this function.
//
// Handler routines for any configured interrupts should be
// added in the appropriate endpoint handler call slots.
//
void USB_ISR () interrupt 8
{
   BYTE bCommonInt, bInInt, bOutInt;

   // Read interrupt registers
   UREAD_BYTE(CMINT, bCommonInt);
   UREAD_BYTE(IN1INT, bInInt);
   UREAD_BYTE(OUT1INT, bOutInt);

   // Check for reset interrupt
```

```
   if (bCommonInt & rbRSTINT)
   {
      // Call reset handler
      USBReset();
   }

   // Check for Endpoint0 interrupt
   if (bInInt & rbEP0)
   {
      // Call Endpoint0 handler
      Endpoint0 ();
   }

   // Endpoint1 IN
   if (bInInt & rbIN1)
   {
      // Call Endpoint1 IN handler
      BulkOrInterruptIn(&gEp1InStatus);
   }

   // Endpoint2 OUT
   if (bOutInt & rbOUT2)
   {
      // Call Endpoint2 OUT handler
      BulkOrInterruptOut(&gEp2OutStatus);
   }
}

//---------------------------
//  Endpoint0 Handler
//---------------------------
void Endpoint0 ()
{
   BYTE bTemp = 0;
   BYTE bCsr1, uTxBytes;

   UWRITE_BYTE(INDEX, 0);                 // Target ep0
   UREAD_BYTE(E0CSR, bCsr1);

   // Handle Setup End
   if (bCsr1 & rbSUEND)                   // Check for setup end
   {                                      // Indicate setup end serviced
      UWRITE_BYTE(E0CSR, rbSSUEND);
      gEp0Status.bEpState = EP_IDLE;      // ep0 state to idle
   }

   // Handle sent stall
   if (bCsr1 & rbSTSTL)                   // If last state requested a stall
   {                                      // Clear Sent Stall bit (STSTL)
      UWRITE_BYTE(E0CSR, 0);
      gEp0Status.bEpState = EP_IDLE;      // ep0 state to idle
   }

   // Handle incoming packet
   if (bCsr1 & rbOPRDY)
   {
      // Read the 8-byte command from Endpoint0 FIFO
      FIFORead(0, 8, (BYTE*)&gEp0Command);
```

```
// Byte-swap the wIndex field
bTemp = gEp0Command.wIndex.c[1];
gEp0Command.wIndex.c[1] = gEp0Command.wIndex.c[0];
gEp0Command.wIndex.c[0] = bTemp;

// Byte-swap the wValue field
bTemp = gEp0Command.wValue.c[1];
gEp0Command.wValue.c[1] = gEp0Command.wValue.c[0];
gEp0Command.wValue.c[0] = bTemp;

// Byte-swap the wLength field
bTemp = gEp0Command.wLength.c[1];
gEp0Command.wLength.c[1] = gEp0Command.wLength.c[0];
gEp0Command.wLength.c[0] = bTemp;

// Decode received command
switch (gEp0Command.bmRequestType & CMD_MASK_COMMON)
{
   case  CMD_STD_DEV_OUT:           // Standard device requests
      // Decode standard OUT request
      switch (gEp0Command.bRequest)
      {
         case SET_ADDRESS:
            SetAddressRequest();
            break;
         case SET_FEATURE:
            SetFeatureRequest();
            break;
         case CLEAR_FEATURE:
            ClearFeatureRequest();
            break;
         case SET_CONFIGURATION:
            SetConfigurationRequest();
            break;
         case SET_INTERFACE:
            SetInterfaceRequest();
            break;
         // All other OUT requests not supported
         case SET_DESCRIPTOR:
         default:
            gEp0Status.bEpState = EP_ERROR;
            break;
      }
      break;

   // Decode standard IN request
   case CMD_STD_DEV_IN:
      switch (gEp0Command.bRequest)
      {
         case GET_STATUS:
            GetStatusRequest();
            break;
         case GET_DESCRIPTOR:
            GetDescriptorRequest();
            break;
         case GET_CONFIGURATION:
            GetConfigurationRequest();
            break;
         case GET_INTERFACE:
```

```
                GetInterfaceRequest();
                break;
             // All other IN requests not supported
             case SYNCH_FRAME:
             default:
                gEp0Status.bEpState = EP_ERROR;
                break;
          }
          break;
       // All other requests not supported
       default:
          gEp0Status.bEpState = EP_ERROR;
    }

    // Write E0CSR according to the result of the serviced out packet
    bTemp = rbSOPRDY;
    if (gEp0Status.bEpState == EP_ERROR)
    {
       bTemp |= rbSDSTL;                // Error condition handled
                                        // with STALL
       gEp0Status.bEpState = EP_IDLE;   // Reset state to idle
    }

    UWRITE_BYTE(E0CSR, bTemp);
}

bTemp = 0;                              // Reset temporary variable

// If state is transmit, call transmit routine
if (gEp0Status.bEpState == EP_TX)
{
    // Check the number of bytes ready for transmit
    // If less than the maximum packet size, packet will
    // not be of the maximum size
    if (gEp0Status.uNumBytes <= EP0_MAXP)
    {
       uTxBytes = gEp0Status.uNumBytes;
       gEp0Status.uNumBytes = 0;        // update byte counter
       bTemp |= rbDATAEND;              // This will be the last
                                        // packet for this transfer
       gEp0Status.bEpState = EP_IDLE;   // Reset endpoint state
    }

    // Otherwise, transmit maximum-length packet
    else
    {
       uTxBytes = EP0_MAXP;
       gEp0Status.uNumBytes -= EP0_MAXP;// update byte counter
    }

    // Load FIFO
    FIFOWrite(0, uTxBytes, (BYTE*)gEp0Status.pData);

    // Update data pointer
    gEp0Status.pData = (BYTE*)gEp0Status.pData + uTxBytes;

    // Update Endpoint0 Control/Status register
    bTemp |= rbINPRDY;                  // Always transmit a packet
                                        // when this routine is called
```

```
                                        // (may be zero-length)

      UWRITE_BYTE(E0CSR, bTemp);        // Write to Endpoint0 Control/Status
   }

}

//---------------------------
// Bulk or Interrupt OUT Handler
//---------------------------
void BulkOrInterruptOut(PEP_STATUS pEpOutStatus)
{
   UINT uBytes;
   BYTE bTemp = 0;
   BYTE bCsrL, bCsrH;

   UWRITE_BYTE(INDEX, pEpOutStatus->bEp); // Index to current endpoint
   UREAD_BYTE(EOUTCSRL, bCsrL);
   UREAD_BYTE(EOUTCSRH, bCsrH);

   // Make sure this endpoint is not halted
   if (pEpOutStatus->bEpState != EP_HALTED)
   {
      // Handle STALL condition sent
      if (bCsrL & rbOutSTSTL)
      {
         // Clear Send Stall, Sent Stall, and data toggle
         UWRITE_BYTE(EOUTCSRL, rbOutCLRDT);
      }

      // Read received packet(s)
      // If double-buffering is enabled, multiple packets may be ready
      while(bCsrL & rbOutOPRDY)
      {
         // Get packet length
         UREAD_BYTE(EOUTCNTL, bTemp);     // Low byte
         uBytes = (UINT)bTemp & 0x00FF;

         UREAD_BYTE(EOUTCNTH, bTemp);     // High byte
         uBytes |= (UINT)bTemp << 8;

         if (uBytes == 8)
         {
            // Read FIFO
            FIFORead(pEpOutStatus->bEp, uBytes, (BYTE*)&OUT_PACKET);
            pEpOutStatus->uNumBytes = uBytes;
         }

         // If a data packet of anything but 8 bytes is received, ignore
         // and flush the FIFO
         else
         {
            UWRITE_BYTE(EOUTCSRL, rbOutFLUSH);
         }

         // Clear out-packet-ready
         UWRITE_BYTE(INDEX, pEpOutStatus->bEp);
         UWRITE_BYTE(EOUTCSRL, 0);
```

```
         // Read updated status register
         UREAD_BYTE(EOUTCSRL, bCsrL);
      }
   }
}

//----------------------------
//  Endpoint Bulk or Interrupt IN Handler
//----------------------------
void BulkOrInterruptIn (PEP_STATUS pEpInStatus)
{
   BYTE bCsrL, bCsrH;

   UWRITE_BYTE(INDEX, pEpInStatus->bEp);  // Index to current endpoint
   UREAD_BYTE(EINCSRL, bCsrL);
   UREAD_BYTE(EINCSRH, bCsrH);

   // Make sure this endpoint is not halted
   if (pEpInStatus->bEpState != EP_HALTED)
   {
      // Handle STALL condition sent
      if (bCsrL & rbInSTSTL)
      {
         UWRITE_BYTE(EINCSRL, rbInCLRDT); // Clear Send Stall and Sent Stall,
                                          // and clear data toggle
      }

      // If a FIFO slot is open, write a new packet to the IN FIFO
      while (!(bCsrL & rbInINPRDY))
      {
         pEpInStatus->uNumBytes = 8;
         pEpInStatus->pData = (BYTE*)&IN_PACKET;

         // Write <uNumBytes> bytes to the <bEp> FIFO
         FIFOWrite(pEpInStatus->bEp, pEpInStatus->uNumBytes,
            (BYTE*)pEpInStatus->pData);

         // Set Packet Ready bit (INPRDY)
         UWRITE_BYTE(EINCSRL, rbInINPRDY);

         // Check updated endopint status
         UREAD_BYTE(EINCSRL, bCsrL);
      }
   }
}

//----------------------------
// USBReset
//----------------------------
// - Initialize the global Device Status structure (all zeros)
// - Resets all endpoints
//
void USBReset ()
{
   BYTE i, bPower = 0;
   BYTE * pDevStatus;

  // Reset device status structure to all zeros (undefined)
   pDevStatus = (BYTE *)&gDeviceStatus;
```

```
   for (i=0;i<sizeof(DEVICE_STATUS);i++)
   {
      *pDevStatus++ = 0x00;
   }

   // Set device state to default
   gDeviceStatus.bDevState = DEV_DEFAULT;

   // REMOTE_WAKEUP_SUPPORT and SELF_POWERED_SUPPORT
   // defined in file "usb_desc.h"
   gDeviceStatus.bRemoteWakeupSupport = REMOTE_WAKEUP_SUPPORT;
   gDeviceStatus.bSelfPoweredStatus = SELF_POWERED_SUPPORT;

   // Reset all endpoints

   // Reset Endpoint0
   gEp0Status.bEpState = EP_IDLE;         // Reset Endpoint0 state
   gEp0Status.bEp = 0;                    // Set endpoint number
   gEp0Status.uMaxP = EP0_MAXP;           // Set maximum packet size

   // Reset Endpoint1 IN
   gEp1InStatus.bEpState = EP_HALTED;     // Reset state
   gEp1InStatus.uNumBytes = 0;            // Reset byte counter

   // Reset Endpoint2 OUT
   gEp2OutStatus.bEpState = EP_HALTED;    // Reset state
   gEp2OutStatus.uNumBytes = 0;           // Reset byte counter

   // Get Suspend enable/disable status. If enabled, prepare temporary
   // variable bPower.
   if (SUSPEND_ENABLE)
   {
      bPower = 0x01;                      // Set bit0 (Suspend Enable)
   }

   // Get ISO Update enable/disable status. If enabled, prepare temporary
   // variable bPower.
   if (ISO_UPDATE_ENABLE)
   {
      bPower |= 0x80;                     // Set bit7 (ISO Update Enable)
   }

   UWRITE_BYTE(POWER, bPower);
}
```

usb_stdreq.c

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_stdreq.c
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Source file for USB firmware. Includes service routine
   for usb standard device requests.

****************************************/

#include "c8051F320.h"
#include "usb_regs.h"
#include "usb_structs.h"
#include "usb_main.h"
#include "usb_desc.h"
#include "usb_config.h"
#include "usb_request.h"

extern code DESCRIPTORS gDescriptorMap;
extern DEVICE_STATUS gDeviceStatus;
extern EP_STATUS gEp0Status;
extern EP_STATUS gEp2OutStatus;
extern EP_STATUS gEp1InStatus;
extern EP0_COMMAND gEp0Command;

BYTE bEpState;
UINT uNumBytes;
PIF_STATUS pIfStatus;

//-----------------------------------------------------------------------
//  Standard Request Routines
//-----------------------------------------------------------------------
//
// These functions should be called when an endpoint0 command has
// been received with a corresponding "bRequest" field.
//
// - Each routine performs all command field checking, and
//   modifies fields of the Ep0Status structure accordingly
//
// After a call to a standard request routine, the calling routine
// should check Ep0Status.bEpState to determine the required action
// (i.e., send a STALL for EP_ERROR, load the FIFO for EP_TX).
// For transmit status, the data pointer (Ep0Status.pData),
// and data length (Ep0Status.uNumBytes) are prepared before the
// standard request routine returns. The calling routine must write
// the data to the FIFO and handle all data transfer

//---------------------------
// SET_ADDRESS device request
//---------------------------
void SetAddressRequest ()
{
   // Index and Length fields must be zero
```

```
   // Device state must be default or addressed
   if ((gEp0Command.wIndex.i) || (gEp0Command.wLength.i) ||
   (gDeviceStatus.bDevState == DEV_CONFIG))
   {
      bEpState = EP_ERROR;
   }

   else
   {
      // Assign new function address
      UWRITE_BYTE(FADDR, gEp0Command.wValue.c[1]);
      if (gDeviceStatus.bDevState == DEV_DEFAULT &&
      gEp0Command.wValue.c[1] != 0)
      {
         gDeviceStatus.bDevState = DEV_ADDRESS;
      }
      if (gDeviceStatus.bDevState == DEV_ADDRESS &&
      gEp0Command.wValue.c[1] == 0)
      {
         gDeviceStatus.bDevState = DEV_ADDRESS;
      }
      bEpState = EP_IDLE;
   }
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// SET_FEATURE device request
//----------------------------
void SetFeatureRequest ()
{
   // Length field must be zero
   // Device state must be configured, or addressed with Command Index
   // field == 0
   if ((gEp0Command.wLength.i != 0) ||
   (gDeviceStatus.bDevState == DEV_DEFAULT) ||
   (gDeviceStatus.bDevState == DEV_ADDRESS && gEp0Command.wIndex.i != 0))
   {
      bEpState = EP_ERROR;
   }

   // Handle based on recipient
   switch(gEp0Command.bmRequestType & CMD_MASK_RECIP)
   {
      // Device Request - Return error as remote wakeup is not supported
      case CMD_RECIP_DEV:
         bEpState = EP_ERROR;
         break;

      // Endpoint Request
      case CMD_RECIP_EP:
         if (gEp0Command.wValue.i == ENDPOINT_HALT)
         {
            bEpState = HaltEndpoint(gEp0Command.wIndex.i);
            break;
         }
         else
         {
            bEpState = EP_ERROR;
```

```
            break;
         }
      default:
         bEpState = EP_ERROR;
         break;
   }
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// CLEAR_FEATURE device request
//----------------------------
void ClearFeatureRequest ()
{
   // Length field must be zero
   // Device state must be configured, or addressed with Command Index
   // field == 0
   if ((gEp0Command.wLength.i != 0) || (gDeviceStatus.bDevState == DEV_DEFAULT) ||
   (gDeviceStatus.bDevState == DEV_ADDRESS && gEp0Command.wIndex.i != 0))
   {
      bEpState = EP_ERROR;
   }

   // Handle based on recipient
   switch(gEp0Command.bmRequestType & CMD_MASK_RECIP)
   {
      // Device Request
      case CMD_RECIP_DEV:
         // Remote wakeup not supported
         bEpState = EP_ERROR;
         break;

      // Endpoint Request
      case CMD_RECIP_EP:
         if (gEp0Command.wValue.i == ENDPOINT_HALT)
         {
            // Enable the selected endpoint.
            bEpState = EnableEndpoint(gEp0Command.wIndex.i);
            break;
         }
         else
         {
           bEpState = EP_ERROR;
           break;
         }
      default:
         bEpState = EP_ERROR;
         break;
   }
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// SET_CONFIGURATION device request
//----------------------------
void SetConfigurationRequest ()
{
   // Index and Length fields must be zero
   // Device state must be addressed or configured
```

```
   if ((gEp0Command.wIndex.i) || (gEp0Command.wLength.i) ||
   (gDeviceStatus.bDevState == DEV_DEFAULT))
   {
      bEpState = EP_ERROR;
   }

   else
   {
      // Make sure assigned configuration exists
      if (gEp0Command.wValue.c[1] >
      gDescriptorMap.bStdDevDsc[std_bNumConfigurations])
      {
         bEpState = EP_ERROR;
      }

      // Handle zero configuration assignment
      else if  (gEp0Command.wValue.c[1] == 0)
         gDeviceStatus.bDevState = DEV_ADDRESS;

      // Select the assigned configuration
      else
         bEpState = SetConfiguration(gEp0Command.wValue.c[1]);
   }
   gEp0Status.bEpState = bEpState;
}


//----------------------------
// SET_INTERFACE device request
//----------------------------
void SetInterfaceRequest()
{
   // Length field must be zero
   if ((gEp0Command.wLength.i) || (gDeviceStatus.bDevState != DEV_CONFIG))
      bEpState = EP_ERROR;

   else
   {
      // Check that target interface exists for this configuration
      if(gEp0Command.wIndex.i > gDeviceStatus.bNumInterf - 1)
         bEpState = EP_ERROR;

      else
      {
         // Get pointer to interface status structure
         pIfStatus = (PIF_STATUS)&gDeviceStatus.IfStatus;

         // Check that alternate setting exists for the interface
         if (gEp0Command.wValue.i > pIfStatus->bNumAlts)
            bEpState = EP_ERROR;

         // Assign alternate setting
         else
         {
            pIfStatus->bCurrentAlt = gEp0Command.wValue.i;
            bEpState = SetInterface(pIfStatus);
         }
      }
   }
```

```
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// GET_STATUS device request
//----------------------------
void GetStatusRequest ()
{
   // Value field must be zero; Length field must be 2
   if ((gEp0Command.wValue.i != 0) || (gEp0Command.wLength.i != 0x02) ||
   (gDeviceStatus.bDevState == DEV_DEFAULT) ||
   (gDeviceStatus.bDevState == DEV_ADDRESS && gEp0Command.wIndex.i != 0))
   {
      bEpState = EP_ERROR;
   }

   else
   {
      // Check for desired status (device, interface, endpoint)
      switch (gEp0Command.bmRequestType & CMD_MASK_RECIP)
      {
         // Device
         case CMD_RECIP_DEV:
            // Index must be zero for a Device status request
            if (gEp0Command.wIndex.i != 0)
               bEpState = EP_ERROR;
            else
            {
               // Prepare data_out for transmission
               gEp0Status.wData.c[1] = 0;
               gEp0Status.wData.c[0] = gDeviceStatus.bRemoteWakeupStatus;
               gEp0Status.wData.c[0] |= gDeviceStatus.bSelfPoweredStatus;
            }
            break;

         // Interface
         case CMD_RECIP_IF:
            // Prepare data_out for transmission
            gEp0Status.wData.i = 0;
            break;

         // Endpoint
         case CMD_RECIP_EP:
            // Prepare data_out for transmission
            gEp0Status.wData.i = 0;
            if (GetEpStatus(gEp0Command.wIndex.i) == EP_HALTED)
               gEp0Status.wData.c[0] |= 0x01;
            break;

         // Other cases unsupported
         default:
            bEpState = EP_ERROR;
            break;
      }

      // Endpoint0 state assignment
      bEpState = EP_TX;

      // Point ep0 data pointer to transmit data_out
```

```
      gEp0Status.pData = (BYTE *)&gEp0Status.wData.i;
      gEp0Status.uNumBytes = 2;
   }
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// GET_DESCRIPTOR device request
//----------------------------
void GetDescriptorRequest ()
{
   WORD wTempInt;

   // This request is valid in all device states
   // Switch on requested descriptor (Value field)
   switch (gEp0Command.wValue.c[0])
   {
      // Device Descriptor Request
      case DSC_DEVICE:
         // Get size of the requested descriptor
         uNumBytes = STD_DSC_SIZE;
         // Prep to send the requested length
         if (uNumBytes > gEp0Command.wLength.i)
         {
            uNumBytes = gEp0Command.wLength.i;
         }
         // Point data pointer to the requested descriptor
         gEp0Status.pData = (void*)&gDescriptorMap.bStdDevDsc;
         bEpState = EP_TX;
         break;

      // Configuration Descriptor Request
      case DSC_CONFIG:
         // Make sure requested descriptor exists
         if (gEp0Command.wValue.c[1] >
         gDescriptorMap.bStdDevDsc[std_bNumConfigurations])
         {
            bEpState = EP_ERROR;
         }
         else
         {
            // Get total length of this configuration descriptor
            // (includes all associated interface and endpoints)
            wTempInt.c[1] = gDescriptorMap.bCfg1[cfg_wTotalLength_lsb];
            wTempInt.c[0] = gDescriptorMap.bCfg1[cfg_wTotalLength_msb];
            uNumBytes = wTempInt.i;

            // Prep to transmit the requested length
            if (uNumBytes > gEp0Command.wLength.i)
            {
               uNumBytes = gEp0Command.wLength.i;
            }
            // Point data pointer to requested descriptor
            gEp0Status.pData = &gDescriptorMap.bCfg1;
            bEpState = EP_TX;
         }
         break;
   }
   gEp0Status.uNumBytes = uNumBytes;
```

```
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// GET_CONFIGURATION device request
//----------------------------
void GetConfigurationRequest ()
{
   // Length field must be 1; Index field must be 0;
   // Value field must be 0
   if ((gEp0Command.wLength.i != 1) || (gEp0Command.wIndex.i) ||
   (gEp0Command.wValue.i) || (gDeviceStatus.bDevState == DEV_DEFAULT))
   {
      bEpState = EP_ERROR;
   }

   else if (gDeviceStatus.bDevState == DEV_ADDRESS)
   {
      // Prepare data_out for transmission
      gEp0Status.wData.i = 0;
      // Point ep0 data pointer to transmit data_out
      gEp0Status.pData = (BYTE *)&gEp0Status.wData.i;
      // ep0 state assignment
      bEpState = EP_TX;
   }

   else
   {
      // Index to desired field
      gEp0Status.pData = (void *)&gDescriptorMap.bCfg1[cfg_bConfigurationValue];

      // ep0 state assignment
      bEpState = EP_TX;
   }
   gEp0Status.uNumBytes = 1;
   gEp0Status.bEpState = bEpState;
}

//----------------------------
// GET_INTERFACE device request
//----------------------------
void GetInterfaceRequest ()
{
   // Value field must be 0; Length field must be 1
   if ((gEp0Command.wValue.i) || (gEp0Command.wLength.i != 1) ||
   (gDeviceStatus.bDevState != DEV_CONFIG))
   {
      bEpState = EP_ERROR;
   }

   else
   {
      // Make sure requested interface exists
      if (gEp0Command.wIndex.i > gDeviceStatus.bNumInterf - 1)
         bEpState = EP_ERROR;
      else
      {
         // Get current interface setting
         gEp0Status.pData = (void *)&gDeviceStatus.IfStatus->bCurrentAlt;
```

```
            // Length must be 1
            gEp0Status.uNumBytes = 1;
            bEpState = EP_TX;
         }
      }
      gEp0Status.bEpState = bEpState;
}
```

usb_utils.c

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:     usb_utils.c
   Author:   JS
   Created:  JAN 03
   Modified: SEP 03 -- FB (FIFORead() - disabled auto read before last byte.)

   Target Device: C8051F320

   Source file for USB firmware. Includes the following support routines:
   - HaltEndpoint()
   - EnableEndpoint()
   - GetEpStatus()
   - SetConfiguration()
   - SetInterface()
   - FIFOWrite()
   - FIFORead()

*****************************************/


#include "c8051F320.h"
#include "usb_regs.h"
#include "usb_structs.h"
#include "usb_main.h"
#include "usb_desc.h"
#include "usb_config.h"
#include "usb_request.h"

extern DEVICE_STATUS gDeviceStatus;
extern code DESCRIPTORS gDescriptorMap;
extern DEVICE_STATUS gDeviceStatus;
extern EP_STATUS gEp0Status;
extern EP_STATUS gEp2OutStatus;
extern EP_STATUS gEp1InStatus;

//----------------------------
// HaltEndpoint()
//----------------------------
//
BYTE HaltEndpoint (UINT uEp)
{
   BYTE bReturnState, bIndex;

   // Save current INDEX value and target selected endpoint
   UREAD_BYTE (INDEX, bIndex);
   UWRITE_BYTE (INDEX, (BYTE)uEp & 0x00EF);

   // Halt selected endpoint and update its status flag
   switch (uEp)
   {
      case EP1_IN:
         UWRITE_BYTE (EINCSRL, rbInSDSTL);
         gEp1InStatus.bEpState = EP_HALTED;
         bReturnState = EP_IDLE;          // Return success flag
         break;
```

```
      case EP2_OUT:
         UWRITE_BYTE (EOUTCSRL, rbOutSDSTL);
         gEp2OutStatus.bEpState = EP_HALTED;
         bReturnState = EP_IDLE;          // Return success flag
         break;
      default:
         bReturnState = EP_ERROR;         // Return error flag
                                          // if endpoint not found
         break;
   }

   UWRITE_BYTE (INDEX, bIndex);           // Restore saved INDEX
   return bReturnState;
}

//---------------------------
// EnableEndpoint()
//---------------------------
//
BYTE EnableEndpoint (UINT uEp)
{
   BYTE bReturnState, bIndex;

   // Save current INDEX value and target selected endpoint
   UREAD_BYTE (INDEX, bIndex);
   UWRITE_BYTE (INDEX, (BYTE)uEp & 0x00EF);

   // Flag selected endpoint has HALTED
   switch (uEp)
   {
      case EP1_IN:
         // Disable STALL condition and clear the data toggle
         UWRITE_BYTE (EINCSRL, rbInCLRDT);
         gEp1InStatus.bEpState = EP_IDLE; // Return success
         bReturnState = EP_IDLE;
         break;
      case EP2_OUT:
         // Disable STALL condition and clear the data toggle
         UWRITE_BYTE (EOUTCSRL, rbOutCLRDT);
         gEp2OutStatus.bEpState = EP_IDLE;// Return success
         bReturnState = EP_IDLE;
         break;
      default:
         bReturnState = EP_ERROR;         // Return error
                                          // if no endpoint found
         break;
   }

   UWRITE_BYTE (INDEX, bIndex);           // Restore INDEX

   return bReturnState;
}

//---------------------------
// GetEpStatus()
//---------------------------
//
BYTE GetEpStatus (UINT uEp)
{
```

```
   BYTE bReturnState;

   // Get selected endpoint status
   switch (uEp)
   {
      case EP1_IN:
         bReturnState = gEp1InStatus.bEpState;
         break;
      case EP2_OUT:
         bReturnState = gEp2OutStatus.bEpState;
         break;
      default:
         bReturnState = EP_ERROR;
         break;
   }

   return bReturnState;
}

//---------------------------
// SetConfiguration()
//---------------------------
//
//
BYTE SetConfiguration(BYTE SelectConfig)
{
   BYTE bReturnState = EP_IDLE;           // Endpoint state return value

   PIF_STATUS pIfStatus;                  // Pointer to interface status
                                          // structure

   // Store address of selected config desc
   gDeviceStatus.pConfig = &gDescriptorMap.bCfg1;

   // Confirm that this configuration descriptor matches the requested
   // configuration value
   if (gDeviceStatus.pConfig[cfg_bConfigurationValue] != SelectConfig)
   {
      bReturnState = EP_ERROR;
   }

   else
   {
      // Store number of interfaces for this configuration
      gDeviceStatus.bNumInterf = gDeviceStatus.pConfig[cfg_bNumInterfaces];

      // Store total number of interface descriptors for this configuration
      gDeviceStatus.bTotalInterfDsc = MAX_IF;

      // Get pointer to the interface status structure
      pIfStatus = (PIF_STATUS)&gDeviceStatus.IfStatus[0];

      // Build Interface status structure for Interface0
      pIfStatus->bIfNumber = 0;           // Set interface number
      pIfStatus->bCurrentAlt = 0;         // Select alternate number zero
      pIfStatus->bNumAlts = 0;            // No other alternates

      SetInterface(pIfStatus);            // Configure Interface0, Alternate0
```

```
      gDeviceStatus.bDevState = DEV_CONFIG;// Set device state to configured
      gDeviceStatus.bCurrentConfig = SelectConfig;// Store current config
   }

   return bReturnState;
}

//---------------------------
// SetInterface()
//---------------------------
// Configure endpoints for the selected interface
//
BYTE SetInterface(PIF_STATUS pIfStatus)
{
   BYTE bReturnState = EP_IDLE;
   BYTE bIndex;

   // Save current INDEX value
   UREAD_BYTE (INDEX, bIndex);

   // Add actions for each possible interface alternate selections
   switch(pIfStatus->bIfNumber)
   {
      // Configure endpoints for interface0
      case 0:
         // Configure Endpoint1 IN
         UWRITE_BYTE(INDEX, 1);           // Index to Endpoint1 registers
         UWRITE_BYTE(EINCSRH, 0x20);      // FIFO split disabled,
                                          // direction = OUT
         UWRITE_BYTE(EOUTCSRH, 0);        // Double-buffering disabled
         gEp1InStatus.uNumBytes = 0;      // Reset byte counter
         gEp1InStatus.uMaxP = EP1_IN_MAXP;// Set maximum packet size
         gEp1InStatus.bEp = EP1_IN;       // Set endpoint address
         gEp1InStatus.bEpState = EP_IDLE; // Set endpoint state

         // Endpoint2 OUT
         UWRITE_BYTE(INDEX, 2);           // Index to Endpoint2 registers
         UWRITE_BYTE(EINCSRH, 0x04);      // FIFO split enabled,
                                          // direction = OUT
         gEp2OutStatus.uNumBytes = 0;     // Reset byte counter
         gEp2OutStatus.uMaxP = EP2_OUT_MAXP;// Set maximum packet size
         gEp2OutStatus.bEp = EP2_OUT;     // Set endpoint number
         gEp2OutStatus.bEpState = EP_IDLE;// Set endpoint state

         // Load first outgoing (IN) packet into FIFO
         BulkOrInterruptIn(&gEp1InStatus);

         UWRITE_BYTE(INDEX, 0);           // Return to index 0

         break;

      // Configure endpoints for interface1
      case 1:

      // Configure endpoints for interface2
      case 2:

      // Default (error)
      default:
```

```
         bReturnState = EP_ERROR;
   }
   UWRITE_BYTE (INDEX, bIndex);           // Restore INDEX

   return bReturnState;
}

//----------------------------
//  FIFO Read
//----------------------------
//
// Read from the selected endpoint FIFO
//
// Inputs:
// bEp: target endpoint
// uNumBytes: number of bytes to unload
// pData: read data destination
//
void FIFORead (BYTE bEp, UINT uNumBytes, BYTE * pData)
{
   BYTE TargetReg;
   UINT i;

   // If >0 bytes requested,
   if (uNumBytes)
   {
      TargetReg = FIFO_EP0 + bEp;         // Find address for target
                                          // endpoint FIFO

      USB0ADR = (TargetReg & 0x3F);       // Set address (mask out bits7-6)
      USB0ADR |= 0xC0;                    // Set auto-read and initiate
                                          // first read

      // Unload <NumBytes - 1> from the selected FIFO
      for(i=0; i<(uNumBytes-1); i++)
      {
         while(USB0ADR & 0x80);           // Wait for BUSY->'0' (data ready)
         pData[i] = USB0DAT;              // Copy data byte
      }

      // Disable auto read and copy last byte
      USB0ADR = (TargetReg & 0x3F);       // Set address (mask out bits7-6)
      while(USB0ADR & 0x80);              // Wait for BUSY->'0' (data ready)
      pData[i] = USB0DAT;                 // Copy final data byte

  }
}

//----------------------------
//  FIFO Write
//----------------------------
//
// Write to the selected endpoint FIFO
//
// Inputs:
// bEp: target endpoint
// uNumBytes: number of bytes to write
// pData: location of source data
//
```

```
void FIFOWrite (BYTE bEp, UINT uNumBytes, BYTE * pData)
{
   BYTE TargetReg;
   UINT i;

   // If >0 bytes requested,
   if (uNumBytes)
   {
      TargetReg = FIFO_EP0 + bEp;         // Find address for target
                                          // endpoint FIFO

      while(USB0ADR & 0x80);              // Wait for BUSY->'0'
                                          // (register available)
      USB0ADR = (TargetReg & 0x3F);       // Set address (mask out bits7-6)

      // Write <NumBytes> to the selected FIFO
      for(i=0;i<uNumBytes;i++)
      {
         USB0DAT = pData[i];
         while(USB0ADR & 0x80);           // Wait for BUSY->'0' (data ready)
      }
   }
}
```

usb_main.h

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_endpoint.h
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Main header file for USB firmware. Includes function prototypes,
   standard constants, device and endpoint state definitions.

*/

#ifndef _USB_MAIN_H_
#define _USB_MAIN_H_

#include "usb_structs.h"

#ifndef _BYTE_DEF_
#define _BYTE_DEF_
typedef unsigned char BYTE;
#endif   /* _BYTE_DEF_ */

#ifndef _WORD_DEF_
#define _WORD_DEF_
typedef union {unsigned int i; unsigned char c[2];} WORD;
#endif   /* _WORD_DEF_ */

// 16-bit SFR declarations
sfr16 DP       = 0x82;                 // data pointer
sfr16 TMR2RL   = 0xca;                 // Timer2 reload
sfr16 TMR2     = 0xcc;                 // Timer2 counter
sfr16 TMR3     = 0x94;                 // Timer3 counter
sfr16 TMR3RL   = 0x92;                 // Timer3 reload
sfr16 PCA0CP1  = 0xe9;                 // PCA0 Module 1 Capture/Compare
sfr16 PCA0CP2  = 0xeb;                 // PCA0 Module 2 Capture/Compare
sfr16 PCA0CP3  = 0xed;                 // PCA0 Module 3 Capture/Compare
sfr16 PCA0CP4  = 0xfd;                 // PCA0 Module 4 Capture/Compare
sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 Capture/Compare
sfr16 PCA0     = 0xf9;                 // PCA0 counter

// Define standard constants
#define TRUE   1
#define FALSE  0
#define ON     1
#define OFF    0
#define REMOTE_WAKE_ON     2
#define REMOTE_WAKE_OFF    0
#define SELF_POWER_ON      1
#define SELF_POWER_OFF     0
#ifndef NULL
#define NULL   0
#endif

#define SYSCLK       23560000            // SYSCLK frequency in Hz
#define SAMPLERATE   100000              // ADC0 Sample Rate
```

```
// USB clock selections (SFR CLKSEL)
#define USB_4X_CLOCK       0
#define USB_INT_OSC_DIV_2  0x10
#define USB_EXT_OSC        0x20
#define USB_EXT_OSC_DIV_2  0x30
#define USB_EXT_OSC_DIV_3  0x40
#define USB_EXT_OSC_DIV_4  0x50

// System clock selections (SFR CLKSEL)
#define SYS_INT_OSC        0
#define SYS_EXT_OSC        0x01
#define SYS_4X_DIV_2       0x02

// USB device speed settings
#define FULL_SPEED         0x20
#define LOW_SPEED          0x00

// ADC0 AMUX settings
#define AMX_TEMP_SENSE     0x1E          // Temperature sensor
#define AMX_P1_7           0x07          // P1.7 (potentiometer)
#define AMX_GND            0x1F          // Ground

// Define endpoint status values
#define  EP_IDLE           0
#define  EP_TX             1
#define  EP_ERROR          2
#define  EP_HALTED         3
#define  EP_RX             4

// Define device states
#define  DEV_DEFAULT       0
#define  DEV_ADDRESS       1
#define  DEV_CONFIG        2

#define  EP_MASK_DIR       0x80

// Endpoint addresses
#define  EP1_IN            0x81
#define  EP1_OUT           0x01
#define  EP2_IN            0x82
#define  EP2_OUT           0x02
#define  EP3_IN            0x83
#define  EP3_OUT           0x03

// Function prototypes
void USBReset (void);                     // usb_reset.c
void Endpoint0 ();                        // usb_endpoint.c
void BulkOrInterruptOut(PEP_STATUS);      //
void BulkOrInterruptIn(PEP_STATUS);       //
void StdReq (PEP_STATUS);                 // usb_stdreq.c
BYTE SetConfiguration(BYTE);              // usb_utils.c
BYTE SetInterface(PIF_STATUS);            //
BYTE HaltEndpoint (UINT uEp);             //
BYTE EnableEndpoint (UINT uEp);           //
BYTE GetEpStatus (UINT uEp);              //
void FIFORead (BYTE, UINT, BYTE*);        //
void FIFOWrite (BYTE bEp, UINT uNumBytes, BYTE * pData);
```

```
// Standard Device Request Routine prototypes
void SetAddressRequest (void);            // usb_stdreq.c
void SetFeatureRequest (void);            //
void ClearFeatureRequest (void);          //
void SetConfigurationRequest (void);      //
void SetDescriptorRequest (void);         //
void SetInterfaceRequest (void);          //
void GetStatusRequest (void);             //
void GetDescriptorRequest (void);         //
void GetConfigurationRequest (void);      //
void GetInterfaceRequest (void);          //
void SynchFrameRequest (void);            //

#endif   /* _USB_MAIN_H_ */
```

usb_config.h

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_config.h
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Header file for usb firmware. Includes device configuration details.
   Constants defined in this file should be modified when changes are
   made to the firmware, including changes to endpoints, interfaces,
   packet sizes, or feature selections (remote wakeup, suspend, etc.).

*/

#ifndef  _USB_CONFIG_H_
#define  _USB_CONFIG_H_

//---------------------------
// Begin device details
//---------------------------
// These constants should be modified to match the current implementation
//
#define NUM_CFG 1                          // Total number of defined
                                           // configurations

#define MAX_IF_DSC 1                       // Maximum number of interface
                                           // descriptors for any defined
                                           // configuration

#define MAX_IF 1                           // Maximum number of interfaces
                                           // for any defined configuration

#define CFG1_IF_DSC 1                      // Total number of interface
                                           // descriptors for configuration1

#define CFG1_EP_DSC 2                      // Total number of endpoint
                                           // descriptors for configuration1

#define REMOTE_WAKEUP_SUPPORT OFF          // This should be "ON" if the
                                           // device is capable of remote
                                           // wakeup (this does not mean that
                                           // remote wakeup is enabled)
                                           // Otherwise "OFF"

#define SELF_POWERED_SUPPORT OFF           // This should be "ON" if the
                                           // device is self-powered;
                                           // "OFF" if the device
                                           // is bus-powered.

#define SUSPEND_ENABLE OFF                 // This should be "ON" if the
                                           // device should respond to suspend
                                           // signaling on the bus.
                                           // Otherwise "OFF"

#define ISO_UPDATE_ENABLE OFF              // This should be "ON" if the ISO
```

```
                                        // Update feature should be turned
                                        // on for all ISO endpoints. It
                                        // should be "OFF" if the ISO
                                        // update feature should not be
                                        // enabled, or if no ISO endpoints
                                        // will be configured

// Endpoint buffer / packet sizes
// These constants should match the desired maximum packet size for each
// endpoint. Note that the size must not exceed the size of the FIFO
// allocated to the target endpoint. This size will depend on the configuration
// of the endpoint FIFO (split mode and double buffer options), as described
// in the device datasheet.
#define EP0_MAXP       64               // Endpoint0 maximum packet size
#define EP1_IN_MAXP    64               // Endpoint1 IN maximum packet size
#define EP2_IN_MAXP    128              // Endpoint2 IN maximum packet size
#define EP3_IN_MAXP    256              // Endpoint3 IN maximum packet size
#define EP1_OUT_MAXP   128              // Endpoint1 OUT maximum packet size
#define EP2_OUT_MAXP   64               // Endpoint2 OUT maximum packet size
#define EP3_OUT_MAXP   256              // Endpoint3 OUT maximum packet size

//---------------------------
// End device details
//---------------------------

#endif   /* _USB_CONFIG_H_ */
```

usb_desc.h

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_desc.h
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Header file for USB firmware. Includes descriptor array index
   definitions.

*/

#ifndef  _USB_DESC_H_
#define  _USB_DESC_H_

// Descriptor sizes
#define STD_DSC_SIZE 18                  // Device
#define CFG_DSC_SIZE 9                   // Configuration
#define IF_DSC_SIZE 9                    // Interface
#define EP_DSC_SIZE 7                    // Endpoint

// Descriptor indicies - these are used to access specific fields
// of a descriptor array.

// Standard Device Descriptor Array Indicies
#define std_bLength 0                    // Length of this descriptor
#define std_bDescriptorType 1            // Device desc type (const. 0x01)
#define std_bcdUSB 2                     // USB Specification used (BCD)
#define std_bDeviceClass 4               // Device Class
#define std_bDeviceSubClass 5            // Device SubClass
#define std_bDeviceProtocol 6            // Device Protocol
#define std_bMaxPacketSize0 7            // Maximum packet size for Endpoint0
#define std_idVendor 8                   // Vendor ID
#define std_idProduct 10                 // Product ID
#define std_bcdDevice 12                 // Device revision number
#define std_iManufacturer 14             // Manufacturer name string index
#define std_std_iProduct 15              // Product name string index
#define std_iSerialNumber 16             // Serial number string index
#define std_bNumConfigurations 17        // Number of supported configurations

// Configuration Descriptor Array Indicies
#define cfg_bLength 0                    // Length of this desc
#define cfg_bDescriptorType 1            // Config desc type (const. 0x02)
#define cfg_wTotalLength_lsb 2           // Total length, including
#define cfg_wTotalLength_msb 3           // interface & endpoints
#define cfg_bNumInterfaces 4             // Number of supported int's
#define cfg_bConfigurationValue 5        // Config index
#define cfg_iConfiguration 6             // Index for string desc
#define cfg_bmAttributes 7               // Power, wakeup options
#define cfg_MaxPower 8                   // Max bus power consumed

// Interface Descriptor Array Indicies
#define if_bLength 0                     // Length of this desc
#define if_bDescriptorType 1             // Interface desc type (const. ??  )
#define if_bInterfaceNumber 2            // This interface index
```

```
#define if_bAlternateSetting 3           // Alternate index
#define if_bNumEndpoints 4               // Endpoints used by this interface
#define if_bInterfaceClass 5             // Device class
#define if_bInterfaceSubClass 6          // Device subclass
#define if_bInterfaceProcotol 7          // Class-specific protocol
#define if_iInterface 8                  // Index for string desc

// Endpoint Descriptor Array Indicies
#define ep_bLength 0                     // Length of this desc
#define ep_bDescriptorType 1             // Endpoint desc type
#define ep_bEndpointAddress 2            // This endpoint address
#define ep_bmAttributes 3                // Transfer type
#define ep_wMaxPacketSize 4              // Max FIFO size
#define ep_bInterval 6                   // Polling interval (int only)

#endif   /* _USB_DESC_H_ */
```

usb_request.h

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_request.h
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Header file for USB firmware. Includes device request
   constants and masks.

*/

#ifndef  _USB_REQUEST_H_
#define  _USB_REQUEST_H_

// Standard Request Codes
#define  GET_STATUS              0x00
#define  CLEAR_FEATURE           0x01
#define  SET_FEATURE             0x03
#define  SET_ADDRESS             0x05
#define  GET_DESCRIPTOR          0x06
#define  SET_DESCRIPTOR          0x07
#define  GET_CONFIGURATION       0x08
#define  SET_CONFIGURATION       0x09
#define  GET_INTERFACE           0x0A
#define  SET_INTERFACE           0x0B
#define  SYNCH_FRAME             0x0C

// bmRequestType Masks
#define  CMD_MASK_DIR            0x80     // Request direction bit mask
#define  CMD_MASK_TYPE           0x60     // Request type bit mask
#define  CMD_MASK_RECIP          0x1F     // Request recipient bit mask
#define  CMD_MASK_COMMON         0xF0     // Common request mask

// bmRequestType Direction Field
#define  CMD_DIR_IN              0x80     // IN Request
#define  CMD_DIR_OUT             0x00     // OUT Request

// bmRequestType Type Field
#define  CMD_TYPE_STD            0x00     // Standard Request
#define  CMD_TYPE_CLASS          0x20     // Class Request
#define  CMD_TYPE_VEND           0x40     // Vendor Request

// bmRequestType Recipient Field
#define  CMD_RECIP_DEV           0x00     // Device Request
#define  CMD_RECIP_IF            0x01     // Interface Request
#define  CMD_RECIP_EP            0x02     // Endpoint Request
#define  CMD_RECIP_OTHER         0x03     // Other Request

// bmRequestType Common Commands
#define  CMD_STD_DEV_OUT         0x00     // Standard Device Request OUT
#define  CMD_STD_DEV_IN          0x80     // Standard Device Request IN
#define  CMD_STD_IF_OUT          0x01     // Standard Interface Request OUT
#define  CMD_STD_IF_IN           0x81     // Standard Interface Request IN
```

```
// Standard Descriptor Types
#define  DSC_DEVICE              0x01     // Device Descriptor
#define  DSC_CONFIG              0x02     // Configuration Descriptor
#define  DSC_STRING              0x03     // String Descriptor
#define  DSC_INTERFACE           0x04     // Interface Descriptor
#define  DSC_ENDPOINT            0x05     // Endpoint Descriptor

#define  DSC_MASK_REMOTE         0x20     // Remote Wakeup Support Mask
                                          // (bmAttributes Config Desc field)

// Feature Selectors (used in set and clear feature commands)
#define  DEVICE_REMOTE_WAKEUP    0x01     // Remote Wakeup selector
#define  ENDPOINT_HALT           0x00     // Endpoint Halt selector

#endif   /* _USB_REQUEST_H_ */
```

usb_structs.h

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_structs.h
   Author:  JS
   Created: JAN 03

   Target Device: C8051F320

   Header file for USB firmware. Includes all data structure definitions.

*/

#include "usb_config.h"
#include "usb_desc.h"

#ifndef  _USB_STRUCTS_H_
#define  _USB_STRUCTS_H_

#ifndef _BYTE_DEF_
#define _BYTE_DEF_
typedef unsigned char BYTE;
#endif   /* _BYTE_DEF_ */

#ifndef _UINT_DEF_
#define _UINT_DEF_
typedef unsigned int UINT;
#endif   /*_UINT_DEF_*/

#ifndef _WORD_DEF_
#define _WORD_DEF_
typedef union {unsigned int i; unsigned char c[2];} WORD;
#endif   /* _WORD_DEF_ */

typedef struct IF_STATUS {
   BYTE bNumAlts;             // Number of alternate choices for this
                              // interface
   BYTE bCurrentAlt;          // Current alternate setting for this interface
                              // zero means this interface does not exist
                              // or the device is not configured
   BYTE bIfNumber;            // Interface number for this interface
                              // descriptor
   } IF_STATUS;
typedef IF_STATUS * PIF_STATUS;


// Configuration status - only valid in configured state
// This data structure assumes a maximum of 2 interfaces for any given
// configuration, and a maximum of 4 interface descriptors (including
// all alternate settings).
typedef struct DEVICE_STATUS {
   BYTE bCurrentConfig;       // Index number for the selected config
   BYTE bDevState;            // Current device state
   BYTE bRemoteWakeupSupport; // Does this device support remote wakeup?
   BYTE bRemoteWakeupStatus;  // Device remote wakeup enabled/disabled
   BYTE bSelfPoweredStatus;   // Device self- or bus-powered
   BYTE bNumInterf;           // Number of interfaces for this configuration
```

```
   BYTE bTotalInterfDsc;      // Total number of interface descriptors for
                              // this configuration (includes alt.
                              // descriptors)
   BYTE* pConfig;             // Points to selected configuration desc
   IF_STATUS IfStatus[MAX_IF];// Array of interface status structures
   } DEVICE_STATUS;
typedef DEVICE_STATUS * PDEVICE_STATUS;

// Control endpoint command (from host)
typedef struct EP0_COMMAND {
   BYTE  bmRequestType;       // Request type
   BYTE  bRequest;            // Specific request
   WORD  wValue;              // Misc field
   WORD  wIndex;              // Misc index
   WORD  wLength;             // Length of the data segment for this request
  } EP0_COMMAND;

// Endpoint status (used for IN, OUT, and Endpoint0)
typedef struct EP_STATUS {
   BYTE  bEp;                 // Endpoint number (address)
   UINT  uNumBytes;           // Number of bytes available to transmit
   UINT  uMaxP;               // Maximum packet size
   BYTE  bEpState;            // Endpoint state
   void *pData;               // Pointer to data to transmit
   WORD  wData;               // Storage for small data packets
   } EP_STATUS;
typedef EP_STATUS * PEP_STATUS;

// Descriptor structure
// This structure contains all usb descriptors for the device.
// The descriptors are held in array format, and are accessed with the offsets
// defined in the header file "usb_desc.h". The constants used in the
// array declarations are also defined in header file "usb_desc.h".
typedef struct DESCRIPTORS {
   BYTE bStdDevDsc[STD_DSC_SIZE];
   BYTE bCfg1[CFG_DSC_SIZE + IF_DSC_SIZE*CFG1_IF_DSC + EP_DSC_SIZE*CFG1_EP_DSC];
} DESCRIPTORS;

#endif   /* _USB_STRUCTS_H_ */
```

usb_regs.h

```
/*
   Copyright 2003 Cygnal Integrated Products, Inc.

   File:    usb_regs.h
   Author:  JS
   Created: JAN 03

   Copyright 2002 Cygnal Integrated Products, Inc.

   Target: C8051F32x
   Tool chain: KEIL C51 6.03 / KEIL EVAL C51

   Header file for USB firmware. Includes all USB core register
   addresses, register access macros, and register bit masks.

*/

#ifndef  _USB_REGS_H_
#define  _USB_REGS_H_

// USB Core Registers
#define  BASE     0x00
#define  FADDR    BASE
#define  POWER    BASE + 0x01
#define  IN1INT   BASE + 0x02
#define  OUT1INT  BASE + 0x04
#define  CMINT    BASE + 0x06
#define  IN1IE    BASE + 0x07
#define  OUT1IE   BASE + 0x09
#define  CMIE     BASE + 0x0B
#define  FRAMEL   BASE + 0x0C
#define  FRAMEH   BASE + 0x0D
#define  INDEX    BASE + 0x0E
#define  CLKREC   BASE + 0x0F
#define  E0CSR    BASE + 0x11
#define  EINCSRL  BASE + 0x11
#define  EINCSRH  BASE + 0x12
#define  EOUTCSRL BASE + 0x14
#define  EOUTCSRH BASE + 0x15
#define  E0CNT    BASE + 0x16
#define  EOUTCNTL BASE + 0x16
#define  EOUTCNTH BASE + 0x17
#define  FIFO_EP0 BASE + 0x20
#define  FIF0_EP1 BASE + 0x21
#define  FIF0_EP2 BASE + 0x22
#define  FIFO_EP3 BASE + 0x23

// USB Core Register Bits

// POWER
#define  rbISOUD        0x80
#define  rbSPEED        0x40
#define  rbUSBRST       0x08
#define  rbRESUME       0x04
#define  rbSUSMD        0x02
#define  rbSUSEN        0x01
```

```
// IN1INT
#define  rbIN3          0x08
#define  rbIN2          0x04
#define  rbIN1          0x02
#define  rbEP0          0x01

// OUT1INT
#define  rbOUT3         0x08
#define  rbOUT2         0x04
#define  rbOUT1         0x02

// CMINT
#define  rbSOF          0x08
#define  rbRSTINT       0x04
#define  rbRSUINT       0x02
#define  rbSUSINT       0x01

// IN1IE
#define  rbIN3E         0x08
#define  rbIN2E         0x04
#define  rbIN1E         0x02
#define  rbEP0E         0x01

// OUT1IE
#define  rbOUT3E        0x08
#define  rbOUT2E        0x04
#define  rbOUT1E        0x02

// CMIE
#define  rbSOFE         0x08
#define  rbRSTINTE      0x04
#define  rbRSUINTE      0x02
#define  rbSUSINTE      0x01

// E0CSR
#define  rbSSUEND       0x80
#define  rbSOPRDY       0x40
#define  rbSDSTL        0x20
#define  rbSUEND        0x10
#define  rbDATAEND      0x08
#define  rbSTSTL        0x04
#define  rbINPRDY       0x02
#define  rbOPRDY        0x01

// EINCSR1
#define  rbInCLRDT      0x40
#define  rbInSTSTL      0x20
#define  rbInSDSTL      0x10
#define  rbInFLUSH      0x08
#define  rbInUNDRUN     0x04
#define  rbInFIFONE     0x02
#define  rbInINPRDY     0x01

// EINCSR2
#define  rbInDBIEN      0x80
#define  rbInISO        0x40
#define  rbInDIRSEL     0x20
#define  rbInFCDT       0x08
#define  rbInSPLIT      0x04
```

```
// EOUTCSR1
#define  rbOutCLRDT     0x80
#define  rbOutSTSTL     0x40
#define  rbOutSDSTL     0x20
#define  rbOutFLUSH     0x10
#define  rbOutDATERR    0x08
#define  rbOutOVRUN     0x04
#define  rbOutFIFOFUL   0x02
#define  rbOutOPRDY     0x01

// EOUTCSR2
#define  rbOutDBOEN     0x80
#define  rbOutISO       0x40

// Register read/write macros
#define UREAD_BYTE(addr,target) USB0ADR = (0x80 | addr); while(USB0ADR & 0x80); target =
USB0DAT
#define UWRITE_BYTE(addr,data) USB0ADR = addr; USB0DAT = data; while(USB0ADR & 0x80)

#endif   /* _USB_REGS_H_ */
```

Notes:

Contact Information

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032
Email: productinfo@silabs.com
Internet: www.silabs.com