Application Report
SPRA910 – June 2003

# A DSP/BIOS USB Device Driver for the TMS320C5509

Software Development Systems

**ABSTRACT**

This document describes the usage and design of the Texas Instruments TMS320C5509 USB peripheral device driver. This device driver is written in conformance to the DSP/BIOS™ IOM device driver model and is USB Specification 1.1 Chapter 9 compliant. Application examples are provided in the DSP/BIOS DDK so users can get started developing applications. By replacing USB descriptors, adding or replacing USB endpoints, and extending or replacing the USB request handlers, one can create a fully functional and customized USB driver. Also included is a brief USB 1.1 specification overview.

**Contents**

## List of Figures

## List of Tables

## 1 Usage

The device driver described here is part of an IOM mini-driver. That is, it is implemented as the lower layer of a 2-layer device driver model. The upper layer is called the class driver and can be either the DSP/BIOS GIO, SIO, or PIP modules. The class driver provides an independent and generic set of APIs and services for a wide variety of mini-drivers and allows the application to use a common interface for I/O requests. Figure 1 shows the overall DSP/BIOS device driver architecture. For more information about the IOM device driver model as well as the GIO, SIO, and PIP modules, see the *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

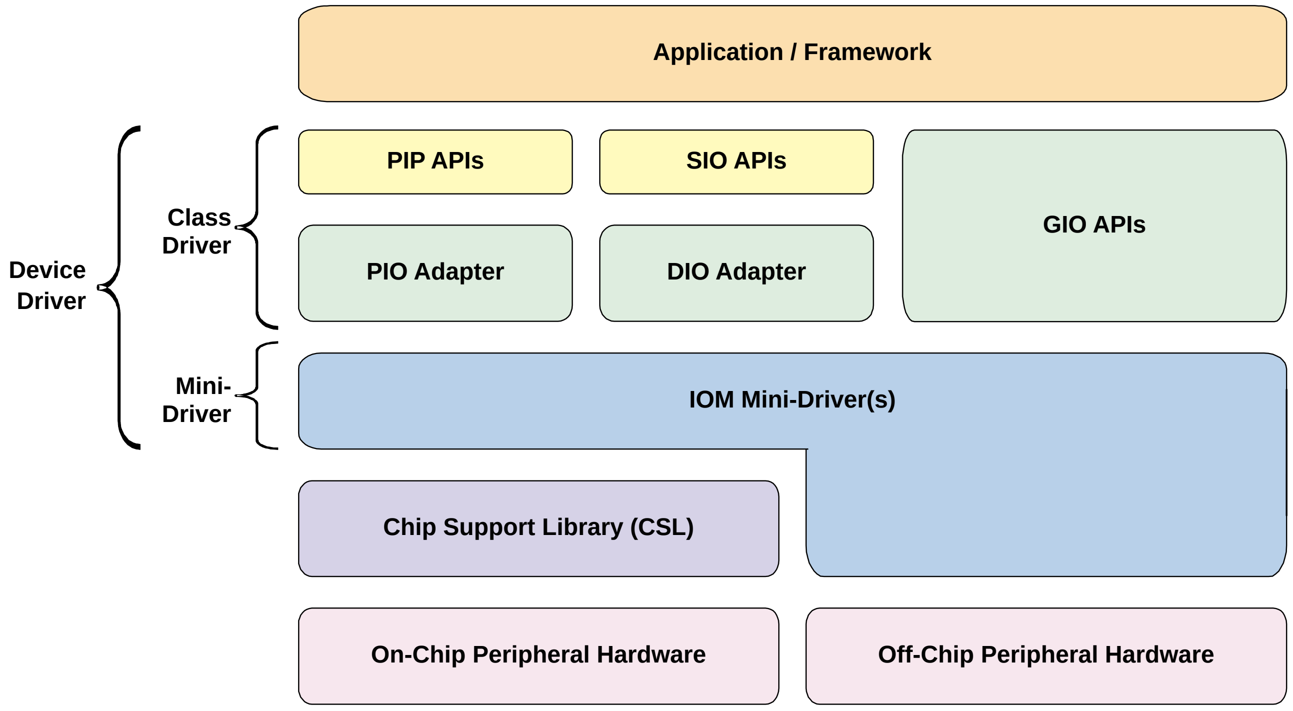

**Figure 1. DSP/BIOS IOM Device Driver Model**

The USB "mini-driver" presented here depends heavily on the Chip Support Library's (CSL) USB module. Please refer to the *TMS320C55x CSL USB Programmer's Reference Guide* (SPRU511) for details on the USB CSL design and usage. This IOM driver calls CSL USB library routines to handle the low level work.

In the rest of this document, all variable, macro, constant and function names in the text will be in *italic*. All data structure types and their fields will be quoted in "*italic*." All the file names will be in **bold**. All the data type and function names prefixed by "USB" are CSL_USB library data type and functions unless otherwise designated.

## 1.1 Configuration

All header files whose names are prefixed with "**c5509_usb_**" define the USB mini-driver's interface and implementation. File **c5509_usb.h** defines the IOM related interface including USB Spec. 1.x Chapter 9 USB request related interface. It allows an application to extend the functionality of the mini-driver built-in default request handler, or even replace the default handler to support vendor specific requests. All data types are prefixed with "C5509_USB".

Mini-driver internal data structures and function prototypes are declared in files prefixed with "_c5509_usb". The file **_c5509_usb.h** defines private data structures and function prototypes.

All mini-driver level source code is contained in C files naming-prefixed with "c5509_usb".

Data types and structures with the "**USB_**" prefix are defined in the CSL header file "**csl_usb.h"**.

## 1.2 Device Parameters

The device parameter structure "*C5509_USB_DevParams*" is defined in file **c5509_usb.h**. The application is required to declare persistent space for the *"C5509_USB_DevParams"* type structure and pass it to function *C5509_USB_mdBindDev()* so that the USB module can be initialized. This structure looks like the following:

```
typedef struct C5509_USB_DevParams
    Uns version;      /* version of mini–driver(e.g. C5509_USB_VERSION0) */
    Uns ier0mask;     /* IER0 mask used by HWI_dispatchPlug in mdBindDev */
    Uns ier1mask;     /* IER1    ″                 ″                    */
    Uns inclk;        /* input clock freq(Mhz at CLKIN pin). See USB_initPLL */
    Uns plldiv;       /* input clock divide down value (CLKIN pin). */
    SmUns pSofTmrCnt; /* 8–bit counter value for pre–SOF timer. See USB_init */
    C5509_USB_DeviceConfig *deviceConfig;  /* device, string & lang id desc */
    C5509_USB_IfcConfig *ifcConfig; /* usb endpoint interface configuration */
} C5509_USB_DevParams;
```

**CAUTION:**

**The USB configuration structures defined by an application must be in persistent memory. The driver does not copy data from this structure but rather references this structure in order to save on overall memory usage.**

- *version* is the mini-driver's revision number. This should be set to the latest revision supported as seen in the **c5509_usb.h** file. If the mini-driver does not support the version specified here, the driver's mdBindDev function will return failure

- *ier0mask* is the IER0 mask used by HWI_dispatchPlug in mdBindDev. Set ier0mask and ier1mask to 1 in order to mask the USB device interrupt itself during ISR processing. Refer to DSP/BIOS HWI module for the usage of these IERx masks.
- *ier1mask* is the IER1 mask used by HWI_dispatchPlug in mdBindDev.
- *inclk* is the input clock freq (MHz at CLKIN pin) used to init the USB CSL PLL. The input clock is divided down using the “plldiv” value below. The CSL function USB_initPLL() uses inclk and plldiv parameters during initialization.
- *plldiv* is the input clock divide down value (CLKIN pin).
- *pSofTmrCnt* is the 8-bit counter value for the pre-start-of-frame timer. This parameter is used for the CSL initialization call “USB_init”.
- *deviceConfig* is the USB device, string and language ID descriptors. See description below.
- *ifcConfig* is the USB endpoint configuration pointer. See description below.

### 1.2.1 C5509_USB_DeviceConfig Structure

The “*C5509_USB_DeviceConfig* “ structure is part of the “*C5509_USB_DevParams*” structure described above and encapsulates USB device specific descriptors, including the USB language ID and string descriptors and USB event handler override functions (“*eventHandler*” and “*setupEventHandler*”). Set “*eventHandler*” and/or “*setupEventHandler*” to null to use default event processing.

```
typedef struct C5509_USB_DeviceConfig {
    Uint16 *deviceDesc;                        /* device descriptor */
    Uint16 *stringDescLangId;                  /* string desc language id */
    String *stringDesc;                        /* string descriptor */
    C5509_USB_eventCb      eventHandler;       /* non–setup event handler */
    C5509_USB_SetupEventCb setupEventHandler;  /* setup event handler */
} C5509_USB_DeviceConfig;
```

- *deviceDesc* is the USB device descriptor with a data format as defined in the USB 1.x Spec Chapter 9.

**NOTE:** Examples of the USB device descriptors can be found in the DSP/BIOS DDK USB example applications driver. The USB descriptor formats are specified in the USB 1.x spec and allocated by the user application on behalf of the USB device driver.

- *stringDescLangId* is the USB language identifier with a data format as defined in the USB 1.x Spec Chapter 9.
- *stringDesc* is the USB string descriptor with a data format as defined in the USB 1.x Spec Chapter 9.
- *eventHandler* is the non-setup event handler function. Set to NULL to use the default handler. When any non-setup event happens, such as the host requesting the device to reset or to suspend, the mini-driver will pass the default event handler function pointer to the application. The application can call this handler function or can override it by providing another handler function.
- *setupEventHandler* is the setup event handler for USB setup events. Set to NULL to use the default handler. If provided, this is the application hook to the USB setup information.

### 1.2.2 C5509_USB_IfcConfig Structure

The “*C5509_USB_IfcConfig*” structure is part of the “*C5509_USB_DevParams*” structure described above and encapsulates the USB endpoint configuration. More than one USB alternate interface may be specified, but only a single USB configuration is supported. Refer to USB 1.x spec for USB descriptor formats.

```
 typedef struct C5509_USB_IfcConfig {
     Uint16                numEps;     /* total number of endpoints */
     USB_DataStruct       *usbConfig;  /* USB config and alt i/f(s) */
     C5509_USB_EpConfig   *epConfig;   /* USB endpoint configuration */
 } C5509_USB_IfcConfig;
```

- “*numEps*” is the number of user-configured endpoints contained in the structure pointed to by epConfig. This number does not include the two reserved control endpoints (EP0 IN and OUT).
- “*usbConfig*” is the list of USB configuration and interface descriptors in the format expected by the underlying USB communications.
- “*epConfig*” is the list of the user USB endpoint descriptors in the format expected by the underlying USB communications.

### 1.2.3 C5509_USB_EpConfig Structure

The “*C5509_USB_EpConfig*” structure is part of the “*C5509_USB_IfcConfig*” structure described above and encapsulates each user endpoint object and its’ configuration.

```
typedef struct C5509_USB_EpConfig {
    USB_EpObj   *eptr;         /* Ptr to CSL unintialized endpoint object */
    USB_EpNum    epNum;        /* Endpoint number (e.g., USB_OUT_EP2) */
    USB_XferType epType;       /* Endpoint transfer type (e.g., USB_BULK, USB_INTR) */
    Uint         epMaxPktLen;  /* Maximum USB packet size(bytes) allowed */
    Uint16       epEvtMask;    /* USB endpoint OR’d event masks(e.g. USB_EVENT_EOT).*/
} C5509_USB_EpConfig;
```

- “*eptr*” is a pointer to a single uninitialized endpoint object. The underlying CSL will initialize all endpoints as part of the mini-drivers’ mdBindDev processing
- “*epNum*” is the USB endpoint number for the associated endpoint (e.g., USB_OUT_EP2).
- “*epType*” is the endpoint type for the associated endpoint. Endpoint type can be USB_CTRL, USB_BULK, USB_INTR or USB_ISO.
- “*epMaxPktLen*” defines the maximum transfer packet length for the associated endpoint. This is the max length of each USB packet transferred, not the max application buffer length.
- “*epEvtMask*” is the associated endpoint event mask containing OR’ed bits representing the USB endpoint events of interest such as an end of posted transaction USB_EVENT_EOT. Refer to the “**csl_usb.h**” file for a complete set of USB events

## 1.3 Channel Parameters

Currently, there is no channel parameter structure used in this driver.

To specify a particular USB endpoint, the user specifies an endpoint by name (e.g., “1”, “2” … “7”) along with the direction, input or output, when creating a IOM channel. For example, for the user to use the configured endpoint USB_IN_EP2, an application using SIO with a configured device named “*usb*” would do something like the following:

```
    outStream = SIO_create(″/usb2 ″, SIO_OUTPUT, SIOBUFSIZE, &attrs);
```

Note the “2” after the device name “*/usb*”. The “2” signifies endpoint 2 and the SIO direction is SIO_OUTPUT (i.e., output from DSP to the host). Note: that a DSP/BIOS “SIO_OUTPUT” stream direction is equivalent a “USB_IN_xxx” direction. An error is returned if the specified endpoint was not configured in the device parameters; see previous section.

## 1.4 USB Event and State Notifications

Several USB device events and states are accessible to DSP applications.

### *1.4.1 Bus Connected Notification*

Before the DSP can initiate transfers on the USB bus, the USB device must be “connected” to the USB host. A typical USB host performs a bus enumeration by identifying newly attached USB device(s) on the bus. The C5509 EVM is identified as such a device and registered with the host software during this USB enumeration phase. As part of the host enumeration the host will send a request to the device, asking for the endpoint interface configuration information. The DSP device driver uses this request from the host as the indication that the host and USB device are now “connected”. The USB device should not attempt to use the bus until this “connected” state is achieved. The USB driver implements a software mechanism to allow DSP application thread(s) to get notified with the bus is connected.

Any DSP thread may make a device driver control call (e.g., SIO_ctrl) with the command code C5509_USB_DEVICECONNECT with an argument pointer to a structure of type *C5509_USB_AppCallback* in order to register a function and argument to be called by the driver when the bus is connected. Refer to the “**c5509_usb.h**” file for structure definitions. For example, if a DSP/BIOS user thread wants to block until the USB bus is connected the application could perform the following:

```
static C5509_USB_AppCallback deviceConnectCb = {

/* function to call when connected */
      (C5509_USB_TappCallback)SEM_post,

/* semaphore handle to post when/if connected */
       usbDeviceConnect

};

/* connect USB device to the host and call deviceConnectCb()  fxn when connected */

SIO_ctrl(outStream, C5509_USB_DEVICECONNECT, &deviceConnectCb );


/* block until bus is connected */

SEM_pend(usbDeviceConnect, SYS_FOREVER);
```

*usbDeviceConnect* is a handle to a semaphore(SEM_Handle).

The user thread will pend on the semaphore if the bus is not connected. Note, the callback(Cb) function will get called immediately if the bus has already been connected.

Another usage scenario that is possible, is to not specify a callback function(e.g., SEM_post), then the application thread will "spin wait" until the bus is connected. The SIO_ctrl() call will only return when the bus is connected. For example, the following call will not return until the bus is connected:

```
/* connect USB device to the host and spin–wait until bus connected */

SIO_ctrl(outStream, C5509_USB_DEVICECONNECT,  NULL ); /* no Cb specified */
```

### 1.4.2 USB State Information (C5509_USB_StateInfo Struct)

Applications can access USB Spec 1.x Chapter 9 internal state information by providing a pointer to a data structure of type *C5509 USB StateInfo* to be filled in by the mini-driver.

```
typedef struct C5509_USB_StateInfo {

    Uint16 usbCurConfig;      /* current USB configuration number */

    Uint16 usbCurIntrfc;      /* current USB interface number */

    Uint16 usbCurAltSet;      /* current USB alternate set number */

    Uint16 usbCurDev;         /* current USB device state */

} C5509_USB_StateInfo;
```

The device specific control command *C5509_USB_GETSTATEINFO* is used along with the *C5509_USB_StateInfo* data structure to retrieve this information. For example, an application can make the following GIO_control() call:

```
    GIO_control(readChan, C5509_USB_GETSTATEINFO, &info);
```

Where *readChan* is any valid IOM channel and *&info* is the address of a *C5509_USB_StateInfo* structure.

Similarly, the SIO API can be used to access this info:

```
    SIO_ctrl(inputStream, C5509_USB_GETSTATEINFO, &info);
```

The contents of the *C5509_USB_StateInfo* info structure is *undefined* if GIO_control() or SIO_ctrl() returns an error.

### 1.4.3 USB Event and Setup Event Handler Override

By default, the IOM USB driver handles the following USB event types as part of the control endpoint 0 processing, host sends USB protocol requests to the C5509 EVM on endpoint 0 OUT(USB_OUT_EP0), and receives the EVM's replies on endpoint 0 IN (USB_OUT_EP0):

- USB_EVENT_RESET
- USB_EVENT_SETUP
- USB_EVENT_SUSPEND
- USB_EVENT_RESUME

- USB_EVENT_EOT

Refer to USB 1.x Chapter 9 for protocol details.

An application can override or extend this event processing by supplying the "*eventHandler*" and/or "*setupEventHandler*" event handlers as part of the *C5509_USB_DeviceConfig* structure in the USB *C5509_USB_DevParams* device configuration structure as described earlier.

The *eventHandler* is a structure of type *C5509_USB_eventCb* that handles the non-setup USB events, such as  USB_EVENT_RESET and USB_EVENT_SUSPEND. An application can override the default *eventHandler* processing as implemented in file **c5509_usb_ctrl.c** by providing a custom handler routine. For example, the function *myEvtCb*() below is an example of an application provided USB non-setup event handler. Since the driver passes the underlying non-setup event handler function to the application, the user can simply extend it or completely override it.

```
static Void myEvtCb( Uint16 event, C5509_USB_UsbEventHandler handler) {

    if (event == USB_EVENT_RESET) {

        /*

         * application can extend functionality here.

         */

        handler();    /* call the default reset event handler */

    }

    else if (event == USB_EVENT_SUSPEND) {

        /*

         * application can extend functionality here

         */

        handler();    /* call the default suspend event handler */

    }

}
```

Similarly, the *setupEventHandler* is a structure of type *C5509_USB_SetupEventCb* that handles the USB setup. An application can gain access to the USB setup data and override or extend the default *setupEventHandler* processing as implemented in files **c5509_usb_reqhndlr.c** and **c5509_usb_ctrl.c** by providing a custom handler routine, for example, the function *mySetupEvtCb*() below shows an application provided USB setup event handler. Since the driver passes a pointer to the USB setup data (*USB_SetupStruct*) as well as the  underlying setup event handler function to the application, the user can simply extend it or completely override it.

```
/*
 * Override default setup event handler
 */
static C5509_USB_UsbReqRet mySetupEvtCb( Uint16 requestId,
        C5509_USB_UsbReqHandler handler, USB_SetupStruct * setupPacket) {
    /*
     * application can extend functionality here.
     */
    return handler();    /* call the default setup event handler */
}
```

Note that the setup event handler returns a status value of type *C5509_USB_UsbReqRet*. This enum type is used to complete the USB transaction by sending the host a USB ACK (_C5509_USB_REQUEST_SEND_ACK, i.e., a zero length data message), getting an ACK from the USB host (_C5509_USB_REQUEST_GET_ACK), stalling the endpoint (_C5509_USB_REQUEST_STALL), or no response at all (_C5509_USB_REQUEST_DONE). Currently, _C5509_USB_REQUEST_DATA_IN and _C5509_USB_REQUEST_DATA_OUT are not used and performs no processing.

```
 typedef enum {
     C5509_USB_REQUEST_DONE = 0,  /* Request done. Can call again after setup */
     C5509_USB_REQUEST_STALL,     /* STALL the control endpoint */
     C5509_USB_REQUEST_SEND_ACK,  /* Send a 0 length IN packet */
     C5509_USB_REQUEST_GET_ACK,   /* Prepare to receive 0 length OUT packet */
     C5509_USB_REQUEST_DATA_IN,   /* Notify handler when IN data transmitted */
     C5509_USB_REQUEST_DATA_OUT   /* Notify handler when OUT data received */
 } C5509_USB_UsbReqRet
```

- USB_REQUEST_DONE is returned to notify the control endpoint 0 handler that the request is completed.
- USB_REQUEST_STALL is returned to notify the control endpoint 0 handler to stall control endpoint USB_OUT_EP0 and USB_IN_EP0.
- USB_REQUEST_SEND_ACK is returned to notify the control endpoint 0 handler to send a 0-byte length IN packet to host.
- USB_REQUEST_GET_ACK is returned to notify the control endpoint 0 handler to prepare to receive a 0-byte length OUT packet from host.
- USB_REQUEST_DATA_IN is returned to notify the control endpoint 0 handler that the host is waiting to receive more data. Currently not used.
- USB_REQUEST_DATA_OUT is returned to notify control endpoint 0 handler that host will continue sending data to device. Currently not used.

This setup event processing is located in file **c5509_usb_ctrl.c** and the user's handler simply needs to return one of the *C5509_USB_UsbReqRet* enum values shown above.

## 2 Architecture

C5509_USB_Fxns is defined in the file **c5509_usb.c**. All IOM mini-driver functions are implemented in files whose names are prefixed with "**c5509_usb_**". The control endpoint 0 handler is implemented in the file **c5509_usb_ctrl.c**. All built-in USB Spec. Chapter 9 request handlers are implemented in file **c5509_usb_reqhndlr.c**.

Figure 2 shows an overview of the USB driver’s salient interfaces and data structures. An important note about the IOM driver architecture is that CSL really performs the major portion of the USB event and data handling. The IOM driver is provided to encapsulate the USB “driver” into the common DSP/BIOS driver model. Refer to the *TMS320C55x CSL USB Programmer’s Reference Guide* (SPRU511).

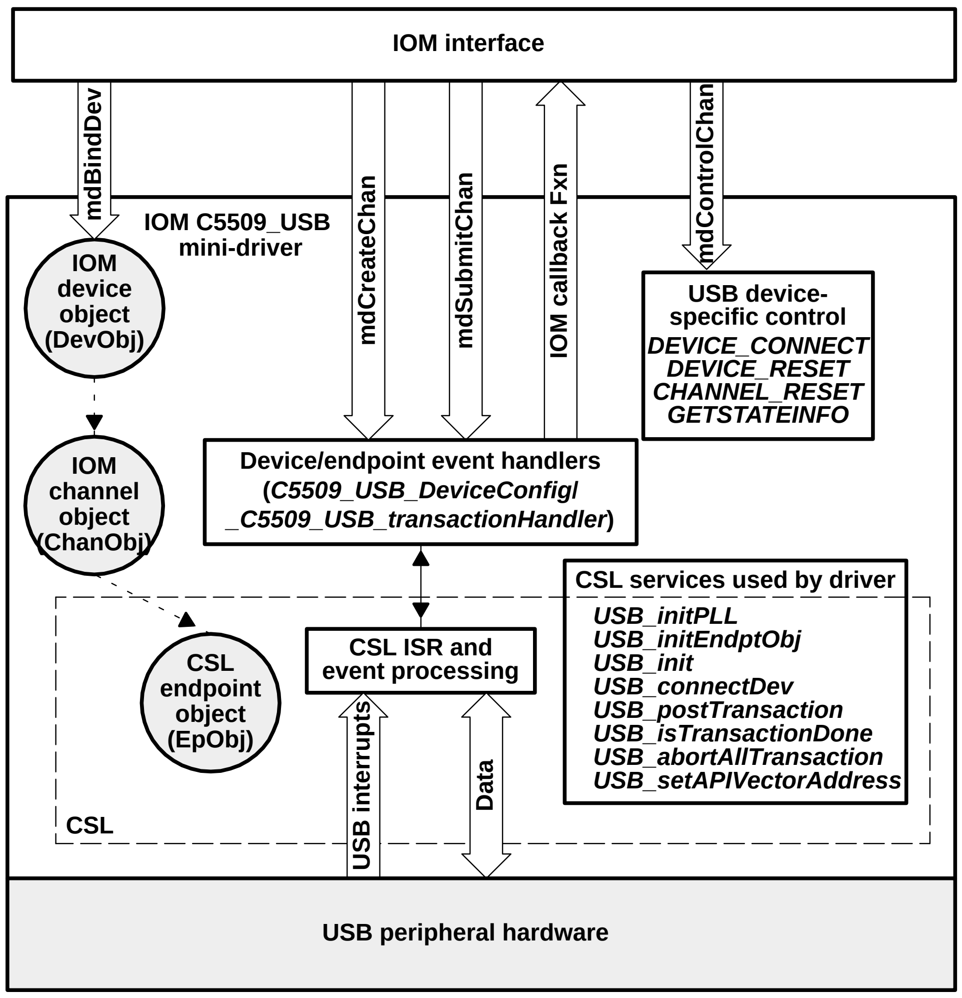

**Figure 2. USB Interfaces and Data Structures**

## 2.1 Internal Data Structures

### *2.1.1 C5509_USB_ChanObj and ChanHandle*

```
typedef struct C5509_USB_ChanObj {
    Uns mode;                    /* IOM_INPUT, IOM_OUTPUT, etc */
    USB_EpHandle endptHandle;    /* endpoint */
    IOM_Packet *flushPacket;     /* IOM_FLUSH/ABORT packet */
    IOM_Packet *dataPacket;      /* current active I/O packet */
    QUE_Obj pendList;            /* list of packets for I/O */
    IOM_TiomCallback cbFxn;      /* IOM callback */
    Ptr  cbArg;                  /* IOM callback argument */
    C5509_USB_TappCallback fxnConnect;  /* Fxn called when bus is connected */
    Ptr argConnect;                     /* argument to fxnConnect() */
} C5509_USB_ChanObj, *ChanHandle;
```

This struct is defined in the public header file **c5509_usb.h** although it is an internally used driver structure. Applications declare all IOM channel objects to avoid dynamically allocating these structures in the driver.

- "*mode*" defines the I/O direction.
- "*endptHandle*" is the pointer to the associated endpoint object.
- "*flushPacket*" is the pointer to the packet handling IOM_flush().
- "*dataPacket*" is the pointer to the current active I/O request packet.
- "*pendList*" holds all pending I/O packets.
- "*cbFxn*" is the pointer to the IOM callback routine.
- "*cbArg*" is the pointer to the IOM callback argument.
- "*fxnConnect*" is the a function pointer to the function type C5509_USB_TappCallback.
- "*argConnect*" is the parameter passed to "function fxnConnect".

### *2.1.2 C5509_USB_DevObj and C5509_USB_DevHandle*

```
typedef struct C5509_USB_DevObj{
    volatile Bool busConnected;   /* Set TRUE after host enumerates bus */
    Uint16  lastRequest;          /* last control channel request */
    C5509_USB_ChanHandle chans[_C5509_USB_ENDPTNUMS]; /* IOM chan ptr array */
    USB_EpHandle eps[_C5509_USB_ENDPTNUMS+1];         /* array of null term'd EPs */
    C5509_USB_StateInfo stateInfo;  /* Internal USB state values */
} C5509_USB_DevObj, *C5509_USB_DevHandle;
extern C5509_USB_DevObj C5509_USB_devObj;
```

This structure is defined in file **_c5509_usb.h**. A global device object, *C5509_USB_devObj* is declared in file **c5509_usb_bind.c** to maintain state and context information for all IOM channels for the device.

The “busConnected” volatile flag is used to record if the host has enumerated the USB bus and requested endpoint descriptor information.

The “lastRequest” field is used to store away the *setupEventHandlers* last USB setup packet.

The “chans[]” array is used to mark a particular channel as being used. An application attempt to open(e.g., SIO_create) the same channel will return and error to the application.

The “*eps*[]”array is the complete private array of sixteen CSL endpoints used to initialize the USB CSL module. The first two endpoint handles belong to the driver to be used as control endpoints. Users may use the remaining 14 endpoints.

### 2.1.3 C5509_USB_UsbRequestStruct

```
typedef struct {
  Uint16 request;    /* request ID */
  C5509_USB_UsbReqHandler usbReqHandler;      /* request handler function pointer */
} _C5509_USB_UsbRequestStruct;
```

This struct is defined in file **_c5509_usb.h** and is used internally to perform a table lookup of the *request* handler given the host request identifier. File **c5509_usb_reqhndlr.c** defines the request-to-handler lookup table *_C5509_USB_usbReqTable*[] used for control endpoint setup event requests.

“request” is the request id. USB Spec. Chapter 9 standard request ID. The “usbReqHandler” is the function pointer to the request handler. The mini-driver supports the following requests:

- _C5509_USB_REQUEST_GET_STATUS – decodes status request from host. Status reported is device status, interface status and endpoint status.
- _C5509_USB_REQUEST_CLEAR_FEATURE – decodes the host’s USB feature request setup packet to install the requested endpoint number or to clear the remote wake-up feature by calling the CSL USB_setRemoteWakeup() function.
- C5509_USB_REQUEST_SET_FEATURE – decodes the host feature request to stall the requested endpoint number or to set the remote wake-up feature by calling the CSL USB_setRemoteWakeup() function.
- _C5509_USB_REQUEST_SET_ADDRESS – Sets the new device address sent in the wValue field of the setup packet by calling the USB_setDevAddr() CSL function.
- _C5509_USB_REQUEST_GET_DESCRIPTOR – Returns the requested USB descriptors.
- _C5509_USB_REQUEST_GET_CONFIGURATION – Return current device configuration value.
- _C5509_USB_REQUEST_SET_CONFIGURATION – Set active configuration of the USB device.
- _C5509_USB _REQUEST_GET_INTERFACE – Return current interface alternate set value.
- _C5509_USB _REQUEST_SET_INTERFACE – Set active interface of the USB device

For other requests, the mini-driver will pass to application the *usbReqUnknown()* routine, which notifies control endpoint 0 handler to stall.

## 2.2 Event Handlers

### *2.2.1 Control Endpoint 0 Handler*

In the file **c5509_usb_ctrl.c**, the function *_C5509_USB_usbCtrlHandler()* is implemented as the USB control endpoint 0 handler. It does the following work:

1. Call *USB_getEvents()* to find out which event happened on endpoint Out0 and In0.
2. If the host requests the device to *RESET* the following is done by default. The *devParams eventHandler* will be NULL, or else it will be the application's responsibility to perform RESET processing:
   - Abort all transactions.
   - Free all pending packets.
   - Set device to default state.
   - Re-configure the USB module by calling *USB_init()*.
   - If the host request SUSPEND, by default the mini-driver performs nothing. The application may override this to, for example, put the device in a power saving state using the *eventHandler* specified in devParams*.*
3. If a setup packet is received, the setup packet is decoded by calling *USB_getSetupPacket()*. If the setup packet is decodable, *USB_lookupReqHandler() is called* to find an appropriate handler to the request and then go to step 5. Otherwise, set *ReqHandlerRet* to *USB_REQUEST_STALL*
4. If the *"setupEventHandler" is specified non-null in devParams* the built-in handler may be overridden for handling setup packets received.
5. Check *ReqHandlerRet* to do the appropriate post processing.
   - If *ReqHandlerRet == C5509_USB_REQUEST_SEND_ACK*, which means the USB device has received all control information from host and the request handler has completed the request. Send a 0-byte IN packet to tell the host that the request is done. Then set *fpRequestHandler* to *USB_reqUnknown*.
   - If *ReqHandlerRet == C5509_USB_REQUEST_GET_ACK*, which means the device already sent to the host the control information requested, set *fpRequestHandler* to *USB_reqUnknown* and prepare to receive a 0-byte OUT packet from the host.
   - If *ReqHandlerRet == C5509_USB_REQUEST_DATA_OUT*, which means device is waiting for more data to get request done, do nothing so that the same request handler will be called next time.
   - If *ReqHandlerRet == C5509_USB_REQUEST_DATA_IN*, which means the device needs to send more control information to the host, nothing is done so that the same request handler will be called again when the current data packet moves out of the endpoint buffer.
   - If *ReqHandlerRet == C5509_USB_REQUEST_STALL*, which means device doesn't understand the setup packet or the request cannot be completed, stall the endpoint Out0 and In0.
   - If *ReqHandlerRet == C5509_USB_REQUEST_DONE*, which means nothing needs to be done, set *fpRequestHandler* to *USB_reqUnknown*.
6. Clear *usbSetup.New* flag so that new setup packets can be detected.

### 2.2.2 Mini-Driver Built-In USB Spec. Chapter 9 Standard Request Handler

- usbReqSetAddress

  This routine calls *USB_setDevAddr()* to set the new USB device address assigned by the host.

- usbReqSetConfiguration

  This function is called when the host requests to set the configuration value. Since only one configuration is supported in this implementation, simply check whether the value is valid. If yes, set *the dev objects usbCurConfig* value assigned by the host and return *C5509_USB_REQUEST_SEND_ACK*. Otherwise, *C5509_USB_REQUEST_STALL* is returned. User needs to extend this function or override this function if multiple configurations are supported.

- usbReqSetInterface

  This function is called when the host requests to set interface. In our example, only one configuration and one interface are supported. We simply check whether the value is valid. If yes, we set *C5509_USB_usbCurAltSetStat* the value assigned by host and return *_C5509_USB_REQUEST_SEND_ACK* so that a 0-byte IN packet will be sent back to host. Otherwise, *_C5509_USB_REQUEST_STALL* is returned to let *_C5509_USB_usbCtrlHandler()* stall the control endpoint Out0 and In0. User needs to extend or override this function if multiple configurations, interfaces are supported.

- usbReqClearSetFeature

  This function is called when host requests to clear/set some features of the device. Currently we only support *USB_FEATURE_REMOTE_WAKEUP* and *USB_FEATURE_ENDPOINT_STALL*. If the request can be handled, a *_C5509_USB_REQUEST_SEND_ACK* is returned. Otherwise, *_C5509_USB_REQUEST_STALL* is returned.

- usbreqGetStatus / usbGetConfiguration / usbGetInterface / usbGetDescriptor

  Device sends appropriate information to the host based on the request and returns *_C5509_USB_REQUEST_GET_ACK* to *USB_ctl()*. Then *_C5509_USB_usbCtrlHandler()* will prepare to receive a 0-byte OUT packet from the host to ensure that the host does receive the data and complete the request. If the request cannot be completed, *_C5509_USB_REQUEST_STALL* is returned.

## 2.3 Mini-Driver Function Implementation

### 2.3.1 C5509_USB_mdBindDev

This function is implemented in the file **c5509_usb_bind.c**. It is not called by application program but rather from DSP/BIOS during initialization. A pre-defined global device parameter, called *devParams* in our example, must be passed to it. It does the following work:

1. Call *USB_setAPIVectorAddress()* to set the USB API vector base address.
2. Initialize USB PLL with the configured devParam structure provided values.
3. Initialize all endpoints. The routine *usbCtrlHandler()* is registered as the event handler for endpoint 0 IN and OUT. The routine *transactionHandler()* is registered as the event handler for other endpoints to handle the actual I/O. In this implementation, most non-control endpoints respond to event EOT only.

4. Call *USB_init()* to initialize the USB module.
5. Set the USB interrupt mask.

The USB device is correctly initiated by now but hasn't been connected to host. To connect it to the host, the global interrupts needs to be enabled so that bus enumeration can start. Since this function is called before main as part of DSP/BIOS initialization when interrupt is not allowed be enabled at this time, we will postpone the work to the point when application calls the device control (e.g., SIO_ctrl() or GIO_control() API) with the *C5509_USB_DEVICECONNECT* command code.

### 2.3.2 C5509_USB_*mdControlChan*

This function is implemented in file **c5509_usb_control.c**. It handles four commands, *IOM_CHAN_RESET*, *IOM_DEVICE_RESET, IOM_CHAN_TIMEDOUT* and *C5509_USB_DEVICECONNECT.* Application can call *GIO_control, for example,* which in turn calls this function to reset a channel or the whole device in case a serious error occurred, or connect the device to the host. Since queue objects need to be manipulated to abort pending I/O packets, the interrupt is disabled at the entry of this function.

- IOM_CHAN_RESET and IOM_CHAN_TIMEDOUT
  - Abort all pending I/Os by calling function *removePackets()* which is defined in file **c5509_usb_submit.c**.
  - Call function *flushPacketHandler()* to complete the GIO_*flush* call if there is any.
  - Restore the interrupt and return *IOM_COMPLETED*.
- IOM_DEVICE_RESET
  - Call function *resetDevice()* which is defined in the same file. *ResetDevice()* will abort all transactions, reset the device which disconnect the device from host, call *freeAllPackets()* to reset all channels, and finally call *freeAllChan()* to free all channels.
  - Restore interrupt.
  - Re-initiate USB module by calling function *reInitUsb()*. *reInitUsb()* set *C5509_USB_usbCurConfigStat* and *C5509_USB_usbCurAltSetStat* to 0, which put the device into default state. It then calls *USB_init()* to init the USB module.
- C5509_USB_DEVICECONNECT
  - Restore interrupt.
  - Call function *connectDevice()* which is defined in the same file. Function *connectDevice()* will call *USB_connectDev()* to pull up D+ wire to notify the USB host that a new device has been attached to the bus.

### 2.3.3 C5509_USB_*mdCreateChan*

This function is implemented in file **c5509_usb_create.c**. *C5509_USB_mdCreateChan()* does the following work:

1. Performs a lookup of the requested endpoint to open by returning the configured channel associated by the named endpoint and direction. If the endpoint can not be found in the user's configured list of endpoints and error is returned to the user.
2. Initialize the channel object and return the channel handle to the user.

### 2.3.4 C5509_USB_mdDeleteChan

This function is not implemented. The mini-driver function table contains the “stub” function IOM_DELETECHANNOTIMPL so any call to mdDeleteChan() will return the IOM_ENOTIMPL error code. This is a memory saving measure since all IOM channel memory is provided by the application when the IOM device driver is “bound” by mdBindDev().

### 2.3.5 C5509_USB_mdSubmitChan

This function is implemented in file **c5509_usb_submit.c**. It handles synchronous I/Os, asynchronous I/Os, flushing and aborting.

- Synchronous I/O (e.g., GIO_read() or GIO_write() API)
  - The upper level class driver (e.g., GIO_submit() function) fills in the I/O request packet and then calls the mini-driver’s C5509_USB_mdSubmitChan() function.
  - The mini-driver *C5509_USB_mdSubmitChan()* appends the I/O packet to the *“pendlist”* if the current I/O has not been completed. Otherwise it posts the current transaction. In either case, *IOM_PENDING* is returned.
  - Seeing that the return value is not *IOM_COMPLETED* for the synchronous I/O, the class drivers GIO_submit() will call *SEM_pend()* to wait for the current I/O to complete. If a timeout occurs, it will call *C5509_USB_mdControlChan()* with the *IOM_CHAN_TIMEOUT* command for the mini-driver to perform any needed timeout processing.
  - When the USB event End-Of-Transaction(*EOT)* occurs, *USB_evDispatch()* finds out which endpoint caused the event to happen, then goes into function *transactionHandler()*. Function *transactionHandle()* will callback into the class driver to notify that an I/O request has been completed.
  - The class driver posts the blocking function (e.g., *SEM_post)* to unblock the calling thread.
  - The class driver *then* returns *IOM_COMPLETED* to the application, indicating the requested I/O is finished.
- Asynchronous I/O

  The application starts asynchronous I/O by directly calling the class drivers *submit() function* with an application callback Fxn. For example for the GIO class driver API it works as follows:
  - *GIO_submit()* tries to get a free packet, fill it, and then call *C5509_USB_mdSubmitChan()*. If no free packets are available, it returns *IOM_ENOPACKETS*.
  - The mini-drivers’ *C5509_USB_mdSubmitChan()* function appends the I/O packet to the *pendlist* if the current I/O has not been completed. Otherwise, it posts the current transaction. In either case, *IOM_PENDING* is returned.
  - *GIO_submit()* simply returns *IOM_PENDING* to application.
  - When event *EOT* occurs, function *transactionHandler*() is called. Function *transactionHandle()* will invoke a callback function to notify the GIO class driver for example that an I/O request is completed.
  - The class driver will put the packet back to the free packet list for later reuse and invoke the application’s callback function to notify application that the I/O is completed.

- Aborting I/O

  Application can directly call *GIO_submit()* with an *IOM_ABORT* command or call *GIO_abort()* to abort all pending I/Os for a channel. GIO*_abort()* is treated as a synchronous request with timeout parameter *SYS_FOREVER*. It works as follows:

  – GIO_submit() fills *syncPacket* and then calls *C5509_USB_mdSubmitChan().*

  – *C5509_USB_mdSubmitChan()* will pass an *IOM_ABORTED* parameter and call *removePackets()* to abort all pending I/Os.

  – For each aborted I/O request, the IOM callback will simply do App. callback with *IOM_ABORTED* status to notify the application that the I/O request is aborted.

  – *C5509_USB_mdSubmitChan()* returns *IOM_COMPLETED* to *GIO_submit()*.

  – *GIO_submit()* returns *IOM_COMPLETED* to the application, indicating that *GIO_abort()* is completed.

- Flushing I/O

  The application can directly call *GIO_submit()* with an *IOM_FLUSH* command or call *GIO_flush()* to flush all pending I/O requests on a channel. *GIO_flush()* is treated as a synchronous request with timeout parameter *SYS_FOREVER*. If the channel is an *IOM_IINPUT* channel, *GIO_flush()* is handled similarly as *GIO_abort(),* except that the status parameter is replaced by *IOM_FLUSHED*.

  Flushing an *IOM_OUTPUT* channel is handled differently, since the IOM specification requires that all pending write requests should be completed routinely. It works as follows for an application using the GIO API :

  – *C5509_USB_mdSubmitChan()* returns *IOM_COMPLETED* if there is no pending write request. In this case, *GIO_submit()* will return *IOM_COMPLETED* to the application, indicating that GIO*_flush() or SIO_idle()* is completed. If there is any pending write request, *flushPacket* is set and *IOM_PENDING* is returned.

  – Seeing that the return value is not *IOM_COMPLETED*, *GIO_submit()* will call *SEM_pend()* to wait forever until the request is complete.

  – When an *EOT* event occurs, the program goes into *transactionHandler()*. It continues to post the next pending transaction until all pending write requests are completed. Then it calls *flushPacketHandler()* to complete the flush request.

  – Function *flushPacketHandler()* will simply set *flushPacket* to NULL and do an OM callback with status *IOM_COMPLETED*.

  – The IOM callback will call *SEM_post()* to unblock the GIO*_submit*() application thread when using tasking.

  – *GIO_submit()* returns *IOM_COMPLETED* to application.

### *2.3.6 C5509_USB_mdUnbindDev*

This function is not implemented. The mini-driver function table contains the “stub” function IOM_BINDDEVNOTIMPL for the mdUnBindDev  table entry to return the IOM_ENOTIMPL error code if this function ever got called.. DSP/BIOS currently never calls mdUnBindDev().

## 3 Constraints

- This device driver and underlying USB CSL currently only supports one USB device.
- This driver is not reentrant for a given IOM channel, which means two different threads cannot issue I/O requests to the same channel in a safe way. It is the application’s responsibility to serialize multithread usage on a single IOM channel.

## 4 References

1. USB Specification 1.1 from http://www.usb.org.
2. *TMS320C55x CSL USB Programmer’s Reference Guide* (SPRU511).
3. *TMS320C55x USB Peripheral Reference Guide* (SPRU317b).
4. *DSP/BIOS Driver Developer’s Guide* (SPRU616).
5. USB host driver and application demo from Thesycon at http://www.thesycon.com.
6. Windows driver programming site: http://www.microsoft.com/ddk/.
7. *Writing Windows Device Drivers: Covers Nt4, Win 98, and Win 2000*, by Chris Cant.
8. *Programming the Microsoft Windows Driver Model*, by Walter Oney.
9. *Writing Linux Device Drivers*, by Takanari Hayama.
10. *Linux Device Drivers*, 2nd Edition, at http://www.xml.com/ldd/chapter/book.
11. *Programming Guide for Linux USB Device Drivers* at http://usb.cs.tum.edu/usbdoc.

# Appendix A Device Driver Data Sheet

## A.1 Device Driver Library Name

C5509_usb.l55 (small memory model) and C5509_usb.l55l (large memory model) for TMS320C5509 DSPs.

## A.2 DSP/BIOS Modules Used

- HWI
- QUE
- IOM

## A.3 DSP/BIOS Objects Used

QUE_Obj

## A.4 CSL Modules Used

- USB
- IRQ

## A.5 CPU Interrupts Used

USB Interrupt IRQ_EVT_USB (bit number 8 in IER0/IFR0)

## A.6 Peripherals Used

- USB

## A.7 Interrupt Disable Time

Maximum time hardware interruptsare disabled by the driver:

- 5,025 cycles

## A.8 Memory Usage

**Table A–1. Device Memory Usage**

| | **Uninitialized Memory** | | **Initialized Memory** | |
|---|---|---|---|---|
| | Small Model | Large Model | Small Model | Large Model |
| **CODE** | — | — | 2970 (8-bit bytes) | 3578 (8-bit bytes) |
| **DATA** | 356 ( 8-bit bytes) | 440 (8-bit bytes) | 314 (8-bit bytes) | 380 (8-bit bytes) |

NOTE: This data was gathered using the sectti command utility.
Uninitialized data: .bss
Initialized data: .cinit + .const
Initialized code: .text + .text:init