MOTOROLA

Wireless System Subscriber Group

Using the 823_USB_API Package to Interface with Your USB Drivers

The purpose of this document is to set the requirement specification and the application programming interface (API) for the *823_USB_API* package.This document is targeted at MPC823 USB device driver programmers and other users of the MPC823 USB controller. The *823_USB_API* package consists of a driver/API to the MPC823 universal serial bus (USB) module and test applications that use the API located on the Motorola Personal Systems website at http://www.mot.com/mpc823.

The *823_USB_API* is a stand-alone product. It provides procedural interfaces, self-contained MPC823 initialization routines, interrupt handling for the USB controller, and example application tests. The software drivers will be used by the MSIL MPC823 design team and, eventually, all MPC823 users.

TERMINOLOGY

The following terms are used throughout this document and defining them may help you to understand the *823_USB_API*.

- Call routine—A routine provided by the *823_USB_API* . For example, Tx_USB_823.
- Callback routine—A routine that should be provided by the user, called a *823_USB_API* interrupt routine. For example, when a USB frame is received the *823_USB_API* will call the user callback routine that will deal with this frame.

If you have hardware platform questions that are not addressed in this document, see the MPC823 specification. You can also refer to the USB specification for additional information on the USB protocol.

FUNCTIONALITY

The *823_USB_API* provides a programming interface for the USB. This includes:

- General MPC823 registers initialization
- Initialization of the USB block
- Interrupt routine(s) for the USB
- USB device routines (set the device address, resume a suspended device)
- Get USB frame (time) number
- Indication of SOF (optional)
- Indication of suspended device and an indication when it resumes
- Indication of busy, reset, and transmit error interrupts
- Indication of received data on an endpoint
- Per endpoint routines—configure, stall or nack an endpoint, ignore in or out tokens, transmit and receive data

- Toggling of data0/data1 for transmit data
- Host routines for debug purposes
- Two example applications that interface to the *823_USB_API*

Note: The *823_USB_API* will be interrupt-oriented. The transmit requests will be issued by the user, but the indications of transmit acknowledge, receive data, and errors will be issued by the driver at a USB interrupt.

The *823_USB_API* does not provide any memory allocation/deallocation mechanism or any transmit/receive error processing.

All these features are the responsibility of the application, which should provide the routines that implement these mechanisms. The application must transmit pointers to these routines as parameters when it calls various *823_USB_API* initialization functions. The example application provided with *823_USB_API* provides some of the above functionality for memory allocation/deallocation and error processing, but it is very limited.

BASIC OPERATION

1. Before operating, the driver must be initialized using the *Init_823* and *Init_USB_823* routine*s*.
2. Any USB physical endpoint should be initialized before its first usage using the routine *Init_USB_Endpoint_823*.
3. At driver initialization, the memory size for Tx & Rx buffer descriptor rings is defined along with the buffer descriptor memory space base address and size. The application has to allocate the memory for the buffer descriptor rings and the driver manages this memory. When each channel is initialized, the driver allocates its Rx & Tx buffer descriptor rings from this memory. The total number of buffer descriptor s for each channel is a function of the memory size for the buffer descriptor rings as defined by the user at initialization and the buffer descriptor size.
4. From there on, frames are transmitted on any initiated endpoint using the *Tx_USB_823* routine.
5. A frame received in an initiated endpoint is either reported by the XX_Call user routine or Rx_USB is immediately called. It is later passed to the user-supplied routine, *f_Store*.
6. The *823_USB_API* operation tries to minimize the number of interrupts generated by the MPC823. Thus, only interrupts per frame transmission / reception are enabled and not interrupts per buffer. Interrupts are disabled whenever possible.

ASSUMPTIONS AND DEPENDENCIES

- There are no dependiencies on the MetaWare High C/C++ Compiler because not all customers will be using it.
- The code should be ANSI C-compliant.
- Some parts of the software were derived from the ATIC software.

REQUIREMENTS

The *823_USB_API* has procedural nterfaces, self-contained MPC823 initialization routines, and interrupt handling.

INTERNAL CAPABILITIES AND DATA STRUCTURES

- data0/data1 Toggling—The *823_USB_API* includes data0/data1 toggling functionality for each one of the four endpoints. For more information about the data0/data1 toggling procedure, see the USB standard.
- USB Device Suspension—A USB device is suspended when it is in an idle state for more then 3ms. *823_USB_API* will maintain an idle timer that is started when an idle on the bus is indicated. This timer will time-out after 3ms and the application program will be notified of the suspension by calling its suspend callback routine. Activity is resumed at the next transmission from the host or by calling the USB_823_Resume routine. When activity resumes, the upper layer will be indicated by calling the exit suspend callback routine.

USB DATA STRUCTURE. The data structure used for the USB and holding routines provided by the user is defined by the following structure:

```
typedef struct {
    e_Err   (* f_Sof)();
                        /* upper layer's routine - used  by */
                        /* the driver to notify the user    */
                        /* of start of frame token.         */
    e_Err   (* f_Tx_Error)(int);
                        /* upper layer's routine - used  by */
                        /* the driver to notify the user    */
                        /* of transmit error.               */
    e_Err   (* f_Busy)();
                        /* upper layer's routine - used  by */
                        /* the driver to notify the user    */
                        /* of 'busy' interrupt.             */
    e_Err   (* f_Reset)();
                        /* upper layer's routine - used  by */
                        /* the driver to notify the user    */
                        /* of 'Reset' interrupt.             */
   e_Err   (* f_Usb_suspend)();
                        /* upper layer's routine - used  by */
                        /* the driver to notify the user    */
                        /* about the suspended device.      */
    e_Err   (* f_Usb_Exit_suspend)();
                        /* upper layer's routine - used  by */
                        /* the driver to notify the user    */
                        /* that that device is no longer    */
                        /*  suspended.                      */
 } t_Usb;
```

ENDPOINT DATA STRUCTURE. The data structure used for each endpoint is defined by the following structure:

```
typedef void *t_Handle;
typedef struct {
      byte              init;       /* 1 if this endpoint was       */
                              /* initialized.               */
      byte              P_Endpoint; /*physical endpoint number  */
                                    /*(0-3).                          */
      t_Handle          RxQ;        /* circular queue for receive         */
                                    /* buffer pool                        */
      t_Handle          ConfQ;      /* frames passed to chip waiting      */
                                    /*for tx                              */
      t_Handle          TxQ;        /* transmit queues                    */
                        crc
      void              *TxBase;    /* first BD in Tx BD table            */
      void              *RxBase;    /* first BD in Rx BD table            */
      void              *ConfBd;    /* next BD for confirm after Tx       */
      void              *TxBd;      /* next BD for new Tx request         */
      void              *RxBd;      /* next BD to collect after Rx        */
      void              *EmptyBd;   /* next BD for new empty buffer       */
      void              *p_RxFrame; /* accumulating receive frame         */
      word              CurrentBuff;/* buffer number within the           */
                                    /* current transmitted frame          */
                                    /*for linking to Tx BD table          */
      ulng              QueueId;    /* id of queue for messages from      */
                                    /* the driver to the upper layer.     */
      e_Err             (* f_Store)(t_Handle,void *);
                        /* upper layer's routine - called to pass         */
                        /* a received frame to the application.           */
      void              *last_bd_of_txed_frame;
                        /* last bd in the last transmitted frame          */
                        /* after  a start transmit command               */
      void              *last_bd_of_ll_frame;
                        /* last bd of last linked frame                   */
      void              *first_bd;
                        /* first bd of the frame currently                */
                        /*being linked                                    */
#ifdef HOST_TEST
      void              *last_bd_of_fl_frame;
                        /* last bd of first linked frame                  */
#endif
      void              *first_bd;
                        /*first bd of the frame currently          */
                        /* being linked.                                 */
      word              data01;
                        /* The actual value of the PID field              */
```

```
                    /* already shifted to the PID field position.*/

} t_Endpoint;

t_Endpoint Endpoint[4];
```

USER-VISIBLE CALL ROUTINES

The *823_USB_API* provides the following user-visible routines that implement the above functionality. All routines return 0 on success. Otherwise, some other negative or positive value is returned.

GENERAL INITIALIZATION. `void Init_823()`

The application should use this call first and only once before any other calls from the *823_USB_API*. This routine only initiates the MPC823 registers/memory that are relevant to USB operation.

USB DEVICE INITIALIZATION. `void Init_USB_823( e_Err (*f_Usb_suspend)(), e_Err (*f_Usb_Exit_suspend)(), e_Err (*f_sof)(), e_Err (*f_Tx_Error)(int Endpoint_Num),e_Err(*f_Busy)(), e_Err(*f_Reset)(),USB_SPECIFIC *param)`

The application has to call this routine to initialize the USB.

Parameters:

- `f_Usb_suspend`—A pointer to the application-provided routine to be called by the *823_USB_API* when an indication of suspend on the USB occurs.
  `f_Usb_Exit_suspend`—A pointer to the application-provided routine to be called by the 823_USB_API when exiting from suspension.
  `f_Sof`- pointer to the application-provided routine to be called by the *823_USB_API* when an indication of SOF frame occurs.
  `f_Tx_Error`—A pointer to the application-provided routine to be called by the 823_USB_API when an interrupt occurs due to a transmission error. The argument passed specifies the endpoint for which the error occurred.
- `f_Busy`—A pointer to the application-provided routine to be called by the 823_USB_API when an interrupt occurs due to a busy error.
- `f_Reset`—A pointer to the application-provided routine to be called by the 823_USB_API when an interrupt occurs due to reset.
- `param`—A pointer to a parameter structure for the USB. See **USB-Specific Parameter Structure** below for more information.

USB-Specific Parameter Structure. For more information on the following code, see the MPC823 USB specification or the USB specification.

```
typedef struct {
      unsigned char     lsp;        /* 0 - 12 Mpbs, 1 - 1.5 Mbps          */
      unsigned char     sof;        /* 0 - no sof indication,             */
                                    /* 1 - sof indication by calling      */
                                    /* user callback routine              */
} USB_SPECIFIC;
```

ENDPOINT INITIALIZATION. `void Init_USB_Endpoint_823(byte Endpoint_Num, ulong x_Bd_Ring_Len,ulong Rx_Bd_Ring_Len, void *DpRam_Base, ulong Bd_Memory_Size, t_Handle *p_Id, ulng QueueId, e_Err (*f_Store)(t_Handle,void *), e_Err (*f_Error)(t_Handle,void *), ENDPOINT_SPECIFIC *param)`

The application has to call this routine to initialize a USB endpoint. For a description of the callback function, `f_Store`, see **USB Endpoint-Specific Parameters Structure** below.

Parameters:

- `Endpoint_Num`—USB physical endpoint number (0-3).
- `Tx_Bd_Ring_Len`—Defines the memory size to be allocated for channel transmit buffer descriptors.
- `Rx_Bd_Ring_Len`—Defines the memory size to be allocated for channel receive buffer descriptors.
- `*DpRam_Base`—Buffer descriptor memory space base.
- `Bd_Memory_Size`—Buffer descriptor memory space size.
- `p_Id`—Handle to the endpoint data structure is returned here.
- `QueueId`—Id of queue for messages from the driver to the upper layer.
- `f_Store`—A pointer to an upper layer routine to be called by the *823_USB_API* to pass received frames.
- `f_Error`—A pointer to an upper layer routine to be called by the *823_USB_API* to pass an error indication.
- `param`—A pointer to a parameter structure for the USB endpoint. See **USB Endpoint-Specific Parameters Structure** below for more information.

USB Endpoint-Specific Parameters Structure. For more information on the following code, see the MPC823 USB specification.

```
typedef enum { Pipe_Bidir = 0, Pipe_Out, Pipe_In } pipe_dir;
typedef enum { Dec = 0, PowerPc, Motorola } byte_ordering;

typedef struct {
      unsigned char     epn;        /* endpoint number 0-15               */
      unsigned char     tm;         /* transfer mode:                     */
                                    /* 0- control, 1-interrupt            */
                                    /* 2-bulk, 3-isochronous              */
      unsigned char     rte;        /* frame retransmit enable:           */
                                    /* 0 - no retransmition               */
                                    /* 1 - automatic retransmission       */
      pipe_dir          dir;        /*  pipe direction:             */
      unsigned short    max_buffer_len/* maximum buffer length            */
      byte_ordering  bo;            /* byte orderring */
} ENDPOINT_SPECIFIC;
```

USB TRANSMIT FRAME. `e_Err Tx_USB_823(t_Handle Ept,void *p_Frame)`

The application calls this function to transmit data frames on a specified endpoint of the USB device. The frame is put in the driver's transmit queue for this endpoint, then an internal routine, *Kick_Tx_USB_823*, is called to handle this frame. See **Kick_Tx_USB_823** below for more information. *Tx_USB_823* takes care of the data0/ data1 field in the transmitted data frame.

Parameters:

- `Ept` - `Handle to endpoint structure`
- `p_Frame` - pointer to frame.

Return value:

- 0 on success . Otherwise, they return some other negative or positive value.

```
e_Err Tx_USB_823(t_Handle Ept, void *p_Frame1)
{
      t_Endpoint *p_Ept = (t_Endpoint *)Ept;

   save interrupt mask status.
   disable transmit interrupts on this endpoint.
   j = CQ_Put(p_Ept->TxQ, p_Frame);  <--- add frame to
                                                          Tx queue
   enable interrupts

   if( j == -1 )              <--- failed to insert frame to Tx queue
   {
      F_Delete( p_Frame );
       return E_FAIL
   } else
            disable transmit interrupt.
            kick the transmitter:
            Kick_Tx_USB_823( p_Ept );
            enable transmit interrupts.
   return E_OK;
}
```

USB HOST TRANSMIT. `e_Err Tx_USB_823(ulong Ept,void *p_Frame)`

The application calls the same function, `Tx_USB_823`(), to transmit a frame on a USB host that will be on endpoint 0. This option is used for testing purposes only. It will be used when the USB is configured to operate in test mode, when this mode endpoint 0 is used as a host, and when the information is looped back to the other three ports. In this mode, the application is responsible for transferring the tokens as well as the data.The prepared token includes the endpoint number of one of the other three physical endpoints and so the packet will be looped to one of those endpoints. This mode is used for transmitting (setup frames, data frames, and SOF frames).

Setup and data will consist of two frames—one for the token and one for the data— which are on a separate buffer descriptor. SOF only consists of the SOF token. In this mode, it is the responsibility of the application to control the data0 and data1 toggling, as well as the retransmission of data when a time-out occurs. Data0/data1 pid are not appended automatically by the USB block (i.e. pid field in the TXBD will be equal to 0).

KICK_TX_USB_823. `static void Kick_Tx_USB_823(t_Endpoint *p_Ept)`

The *Kick_Tx_USB_823* routine passes the frames for transmission to the chip. It is called from the *Tx_USB_823* function when a frame in inserted to an empty transmit queue to put as many buffers as possible in the specified channel's transmit buffer descriptor table. After this, further calls to *Kick_Tx_USB_823* are made from the transmit interrupt routine until all data has been handled. *Kick_Tx_USB_823* also releases previously transmitted frames from the buffer descriptor table and frees their memory.

`Kick_Tx_USB_823` is also used for transmitting frames over a host endpoint. Since HOST mode is intended to be used only by the MSIL MPC823 design team, the compilation flag HOST_TEST is used to differentiate between the testing version of the routine and the normal operation version.

Parameters:

- `p_Ept`—A pointer to the endpoint handler data structure.

```
static void Kick_Tx_USB_823( t_Endpoint *p_Ept )
{

   /* ---------------------------------------- */
   /* collect transmitted BD's from the chip    */
   /* ---------------------------------------- */
    bd = p_Ept->ConfBd;
    while( (!(BD_STATUS(bd) & T_R)) && (BD_BUFFER(bd)) )
    {
      if( last BD in frame )
      {
         p_F = CQ_Get( p_Ept->ConfQ );
          /* If there is no error in the frame, i.e, it has  */
          /* been acknowlendged, toggle the data01 field of  */
          /* the endpoint.                                   */
#ifndef HOST_TEST
          /* If there is no error in the frame, i.e, it has  */
          /* been acknowlendged, toggle the data01 field of  */
          /* the endpoint.                                   */
          if( ! (BD_STATUS(bd) &(TO|UN)))
              p_Ept->data01 = (p_Ept->data01 + 1)&1;
#endif HOST_TEST
          /* If there is an error in the frames last BD, call*/
          /* The user supplied error routine, with the type    */
          /* error and pointer to the frame and only after the */
          /* application routine is called delete the frame    */
          p_Ept->f_Error( p_Ept->Upper, p_F );
         F_Delete( p_F );
      }
```

```
   /* prepare BD for next time */
   BD_BUFFER(bd)  = 0;
   BD_STATUS(bd) &= T_W;
   if( bd == p_Ept->last_bd_of_txed_frame )
   {
   advance BD pointer.
   /* If there is a whole frame awating to be transmitted*/
    /* set the ready bit of the current BD. The CPM      */
     /* will start trnasmition of this BD at the next     */
     /* start transmit command.                           */
     if(p_Ept->last_bd_of_ll_frame)
            bd |= T_R;
      p_Ept->last_bd_of_txed_frame = 0;
     break;
   }
   else
   {
    advance BD pointer.
    }
}

p_Ept->ConfBd = bd;


/* ---------------------------------------- */
/* push as many BD's to the chip as possible */
/* ---------------------------------------- */

i = p_Ept->CurrentBuff;  <--- index of the current buffer within
                                  the frame to be transmitted.
bd = p_Ept->TxBd;  <---  next bd to be used for transmitting
for(;;)
{
   get next frame (p_F) from the transmit queue,
   if the queue is empty go out of the for loop.

   p_B = F_GetBuffer(p_F, i);
   for each buffer in the frame    <-- p_ B != NULL
   {
         if  next BD is not free
               goto TxOut

         set up buffer descriptor
         (pointer & length).
```

```
          /* If the appended buffer is the first buffer     */
          /* in the bd do not set the ready bit and save a  */
           /* pointer to that BD.                            */
           /* The ready bit in that BD will be set later,    */
            /* when all the frame is linked and if it is not  */
            /* the first BD awating to be transmitted after a */
            /* start transmit command to the USB              */
            if( i == 0 )
           {
#ifdef HOST_TEST
                if( p_Ept->P_Endpoint == 0 )
                    if(! p_Ept->first_bd )
                         p_Ept->first_bd = bd;
            }
            else
#endif
                  p_Ept->first_bd = BD;
            }
            else
               BD_STATUS(BD)|= T_R;
         /* set pid field according to the data01 toggle */
         /* and toggle the data01 field of the endpoint  */

         /* see if this is the last buffer in the frame structure */
         if( last buffer in frame)
         {
            i = 0;
            set T_L and T_I bit in bd status
#ifdef HOST
              if( p_Ept->P_Endpoint == 0 )
              {
                  if( F_GetInfo1( p_F ) ) & SET_HOST_LAST )
                       set T_LL bit (that is, HOST LAST);
                  if it is a data frame set crc bit in the bd to 1.
              }
              else
#endif
              {
                  set pid field according to the data01 toggle
                  set crc bit in the bd to 1 /* append crc       */
              }

            put frame in confirmation queue :
            CQ_Put( p_Ept->ConfQ, p_F );

            remove frame from transmit queue :
            CQ_Get( p_Ept->TxQ );
```

```
#ifdef HOST_TEST
            if( p_Ept->P_Endpoint == 0 )
            {
               If there id no ready frame for transmission
               and there is no transmission at the moment,
               set the ready bit on the first BD in the frame.
            }
            else
#endif
            /* Set the ready bit of the first frame, if it is */
            /* not the first bd to be transmitted after a new */
            /* start transmit command.                        */
              if( p_Ept->last_bd_of_txed_frame )
              {
                  if( p_Ept->first_bd !=
                                    (p_Ept->last_bd_of_txed_frame+1bd))
                 BD_STATUS(first_bd) |= T_R;
              }
              else
               BD_STATUS(first_bd) |= T_R;

#ifdef HOST
           if( p_Ept->P_Endpoint == 0 )
              /* for host endpoint, last_bd_of_ll_frame will be*/
              /* set to the current bd only if HOST_LAST is    */
              /* set on that BD                                */
          else
#endif
             p_Ept->last_bd_of_ll_frame = bd;

         }
          else
         {
             if( i != 0 )
                 BD_STATUS_SET(bd, BD_STATUS(bd) | T_R);
             i++;
         }
ó         p_B = next buffer within the frame.
          get next buffer descriptor:
          bd = next bd
      }end - for each buffer in the frame
   }end - for

TxOut:
#ifdef HOST
           if( p_Ept->P_Endpoint == 0 )
            {
```

```
                transmit only one frame.
            }
            else
#endif
        /* If there is a whole frame awating to be transmitted*/
         /* and there is no frame currently being transmitted, */
        /* call the start transmit command.                   */
        if( (p_Ept->last_bd_of_ll_frame)&&
                                  (!p_Ept->last_bd_of_txed_frame) )
        {
            p_Ept->last_bd_of_txed_frame =
                                       p_Ept->last_bd_of_ll_frame;
           p_Ept->last_bd_of_ll_frame = 0;
           start_transmit(p_Ept);
        }

      save parameters for next call:
     p_Ept->TxBd = bd    <--- next bd to be used for transmitting.
   p_Ept->CurrentBuff = i;  <--- current buff index within the frame;
}
```

RECEIVE FRAME. `void Rx_USB_823( t_Endpoint *p_Ept )`

This function is called by the upper application to initiate reception handling. When an interrupt on receive occurs, Rx_USB_823 is either called directly or by the upper application that is notified using the XX_Call routine. The QueueId value determines which of these is chosen. The *Rx_USB_823* routine collects the frames from the endpoint buffer descriptors and passes them to the application by calling another upper application's routine that is pointed by p_Ept->Store. Before notifying the upper layer on the Rx interrupt, all interrupts that occur while receiving frames on this channel are disabled until leaving the *Rx_USB_823* routine.

Parameters:

- ❑ p_Ept—A pointer to the endpoint handler data structure.

```
void Rx_USB_823( t_Endpoint *p_Ept )

{
   t_Handle  CQ = p_Ept->RxQ;
   byte     *bd;
   void     *p_F;
   int       n;
   tBuffer           *p_B;

   /*
      collect received buffers
    */
   p_F = p_Ept->p_RxFrame;
   bd = p_Ept->RxBd;
```

```
RxLoop:
   while( there are received buffers )
   {
      /* get buffer structure associated with this BD */
      p_B = CQ_Get( CQ ); <---  get buffer structure associated
                                        with this BD
      if( p_B )
      {
         B_SetLength( p_B, BD_LENGTH(bd) );

          attach the received buffer to the accumulating frame:
         if( !p_F )
            p_F = F_New(p_B);
         else
         {
            if( first in frame )
            {
                /* if we're starting a new frame before the
                   previous one finished, discard the old one
                   and try again (we're busy)*/

#if DEBUG_LEVEL == 1
                  XX_Event(EV_RECEIVE_DISCARD, p_Ept->P_Endpoint );
#endif
               F_Delete( p_F );
               p_F = F_New(p_B);
            } else
                F_PutBuffer( p_F, p_B );
         }

         /* if end of frame pass up to the user
          */
         if( p_F )
         {
             if( first in frame )
                 F_set_Info1( p_F, BD_STATUS(BD) & PID );
           if( last in frame )
            {
               if( error in frame )
               {
#if DEBUG_LEVEL == 1
                  XX_Event(EV_RECEIVE_FRAME_ERROR,p_Ept->P_Endpoint);
#endif
                  F_Delete(p_F);
                  p_F=0;
                  goto NextBD;
```

```
            }

                Strip crc out of the frame. set length of last
                buffer and of the whole frame according to length
                of CRC striped out of the frame.
                p_Ept->f_Store( p_Ept->Upper, p_F );
              p_F = 0;
           }   <-- end - if(last in frame)
        } else         <-- end - if(p_F)
           B_Delete( p_B, TRUE );
     }   <-- end - if(p_B)
#if DEBUG_LEVEL == 2
     else
           XX_Event(EV_UNEXPECTED_EMPTY_RXQ, p_Ept->P_Endpoint );
#endif

NextBD:
     /* clear the BD for next time */
    BD_BUFFER(bd)  = 0;
     BD_STATUS(bd) &= R_W ;

     advance BD pointer:
      bd = next bd
     Endpoint_FillRxPool( p_Ept );
  }

  p_Ept->RxBd = bd;
  p_Ept->p_RxFrame = p_F;

  /* replenish the receive buffer pool */
  Endpoint_FillRxPool( p_Ept );
}
```

ENDPOINT_FILLRXPOOL. `static void Endpoint_FillRxPool( t_Endpoint *p_Ept )`

This function is called from the *Rx_USB_823* function. It replenishes the receive buffer pool and passes as many empty buffers to the chip as possible.

Parameters:

- p_Ept—A pointer to the endpoint handler data structure.

INTERRUPT ROUTINES. `usb_interrupt()`

The application must call the `usb_interrupt` routine when a USB interrupt occurs. This means that the application has to have its own hardware interrupt detection mechanism. `usb_interrupt` will call a reception interrupt routine, which is responsible for calling, either directly or through XX_Call, the Rx_USB_823. When a transmission interrupt occurs, `usb_interrupt` will call the a transmission interrupt routine, which will in turn call the `Kick_Tx_USB_823` routine.

The application-provided interrupt handlers of the USB interrupt should execute in the following sequence:

1. Save the appropriate registers.
2. Call the `mpc823_Intr`() routine, which will call `usb_interrupt` if a USB interrupt occurred.
3. Store the previously saved registers.
4. Issue the RFI (return from interrupt) command.

An example of how to do this with the MetaWare compiler will be provided in the *823_USB_API* in a file called *inter_low.S*.

Parameters:

- None.

RESUME THE USB DEVICE. `void USB_823_Resume(ulng msec)`

This routine causes the MPC823 to resume a previously suspended device.

Parameters:

- `msec` —An indication of the period of time the request to resume will last. It is recommended that you use 20msec for a host and 10-15msec for a device.

STALL AN ENDPOINT. `void Endpoint_USB_823_Stall_Tx(int Endpoint_Num)`

`void Endpoint_USB_823_Stall_Rx(int Endpoint_Num)`

These routines cause an endpoint to force a STALL handshake for the received/transmitted tokens.

Note: It is the responsibility of the application to remember the state of the pipe before calling this routine to set the pipe back to its normal operation when the stall state is cleared. This can be done by calling one of the routines in— Bidirectional operation for an endpoint, In operation for an endpoint, and Out operation for an endpoint.

Parameters:

- `Endpoint_Num`—Number of physical endpoints (0-3).

NACK AN ENDPOINT. `void Endpoint_USB_823_Nack_Tx(int Endpoint_Num)`

`void Endpoint_USB_823_Nack_Rx(int Endpoint_Num)`

These routines cause an endpoint to force a NACK handshake for the received/transmitted tokens.

Note: It is the responsibility of the application to remember the state of the pipe before calling this routine to set the pipe back to its normal operation when the stall state is cleared. This can be done by calling one of the routines in— Bidirectional operation for an endpoint, In operation for an endpoint, and Out operation for an endpoint.

Parameters:

- `Endpoint_Num`—Number of physical endpoints (0-3).

BIDIRECTIONAL OPERATION FOR AN ENDPOINT. `void Endpoint_USB_823_Bidir(int Endpoint_Num)`

This routine causes an endpoint to send ACK handshake for IN and OUT received tokens.

Parameters:

- `Endpoint_Num`—Number of physical endpoints (0-3).

IN OPERATION FOR AN ENDPOINT. `void Endpoint_USB_823_In(int Endpoint_Num)`

This routine causes an endpoint to send ACK handshake only for IN received tokens.

Parameters:

- `Endpoint_Num`—Number of physical endpoints (0-3).

OUT OPERATION FOR AN ENDPOINT. `void Endpoint_USB_823_Out(intEndpoint_Num)`

This routine causes an endpoint to send ACK handshake only for OUT received tokens.

Parameters:

- `Endpoint_Num`—Number of physical endpoints (0-3).

SET DEVICE ADDRESS (0-127). `void SetAddress_USB_823(unsigned char address)`

This routine sets the address of the device.

Parameters:

- `address`—The device address (0-127).

GET USB FRAME (TIME) NUMBER. `ulng USB_823_get_frame_num()`

This routine returns the frame number as stored at the MPC823 parameter RAM FRAME_N parameter.

GET ENDPOINT PHYSICAL NUMBER. `ulng Endpoint_Get_P_Endpoint ( t_Handle Ept);`

This routine returns the physical endpoint number for a given endpoint structure.

Parameters:

- `Ept`—A pointer to a handler of an endpoint structure.

APPLICATION TEST PROGRAMS

The application test programs can be used for testing purposes and are provided as an example to MPC823 users. Both test programs will initialize the USB to operate in test mode (physical endpoint 0 will operate as a host endpoint and the other three physical endpoints will operate as device endpoints).

GENERAL APPLICATION TEST. The general application test can be found at app_test.c and it performs the following functions:

1. The USB device operates in loop-back mode.
2. Endpoint 1 transmits frames of data to endpoint 0.
3. The number of frames, their length, and data are read from the input files, sw.frame_num_length and sw.input.
4. Endpoint 0 is configured as host and transmits IN TOKENs to endpoint 1.
5. Each IN TOKEN is sent only after it is confirmed that there is a data frame ready for transmission at the FIFO of endpoint 1.
6. When a data frame is received at endpoint 0, it is transmitted and preceded by an OUT TOKEN to endpoint 2.
7. After all frames are received at endpoint 2, their data is printed out to sw.output.
8. The program ends when all transmitted frames from endpoint 1 are received at endpoint 2 and printed out.

IDLE TIMER TEST. The idle timer test can be found at app_test_timer.c and it performs the following functions:

1. The USB device operates in loop-back mode.
2. Endpoint 0 is configured as the host and first transmits five SOF TOKENs.
3. After each transmission, it starts a 4ms timer, which means an idle longer than 3ms. This causes the device to suspend and the f_Usb_suspend callback routine to be called.
4. The device resumes action when the next SOF TOKEN is transmitted and the f_Usb_Exit_suspend callback routine is called.
5. After the fifth suspension, no SOF TOKEN is sent. Instead, USB_823_Resume is called to reactivate the device.
6. The number of calls to f_Usb_suspend and f_Usb_Exit_suspend is counted and printed. This program also tests the USB_823_get_frame_num routine. The frame number as saved in the parameter RAM FRAME_N is printed at this stage, then a different SOF TOKEN is sent and the frame number is printed again.