Microsoft

Azure RTOS
USBX Host Stack
Supplemental User Guide

Published: February 2020

For the latest information, please see
azure.com/rtos

Part Number: 000-1011

Revision 6.0

Chapter 1: Introduction to the USBX Host Stack User Guide Supplement

This document is a supplement to the USBX Host Stack User Guide. It contains documentation for the uncertified USBX Host classes that are not included in the main user guide.

Chapter 2: USBX Host Classes API

This chapter covers all the exposed APIs of the USBX host classes. The following APIs for each class are described in detail:

Printer class
Audio class
Asix class
Pima/PTP class
Prolific class
Generic Serial class

ux_host_class_printer_read

Read from the printer interface

Prototype

```
UINT  ux_host_class_printer_read(UX_HOST_CLASS_PRINTER *printer,
                                 UCHAR *data_pointer,
                                 ULONG requested_length,
                                 ULONG *actual_length)
```

Description

This function reads from the printer interface. The call is blocking and only returns when there is either an error or when the transfer is complete. A read is allowed only on bi-directional printers.

Parameters

printer	Pointer to the printer class instance.
data_pointer	Pointer to the buffer address of the data payload.
requested_length	Length to be received.
actual_length	Length actually received.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported because the printer is not bi-directional.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, reading incomplete.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_printer_read(printer, data_pointer,
                                    requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_write

Write to the printer interface

Prototype

```
UINT  ux_host_class_printer_write(UX_HOST_CLASS_PRINTER *printer,
                                  UCHAR *data_pointer, ULONG requested_length,
                                  ULONG *actual_length)
```

Description

This function writes to the printer interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

printer	Pointer to the printer class instance.
data_pointer	Pointer to the buffer address of the data payload.
requested_length	Length to be sent.
actual_length	Length actually sent.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, writing incomplete.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_printer_write(printer, data_pointer,
                                     requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_soft_reset

Perform a soft reset to the printer

Prototype

```
UINT  ux_host_class_printer_soft_reset(UX_HOST_CLASS_PRINTER *printer)
```

Description

This function performs a soft reset to the printer.

Input Parameter

printer	Pointer to the printer class instance.

Return Values

UX_SUCCESS	(0x00)	The reset was completed.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, reset not completed.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_printer_soft_reset(printer);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_status_get

Get the printer status

Prototype

```
UINT  ux_host_class_printer_status_get(UX_HOST_CLASS_PRINTER *printer,
                                       ULONG *printer_status)
```

Description

This function obtains the printer status. The printer status is similar to the LPT status (1284 standard).

Parameters

printer	Pointer to the printer class instance.
printer_status	Address of the status to be returned.

Return Values

UX_SUCCESS	(0x00)	The reset was completed.
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to perform the operation.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, reset not completed

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_printer_status_get(printer, printer_status);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_read

Read from the audio interface

Prototype

```
UINT  ux_host_class_audio_read(UX_HOST_CLASS_AUDIO *audio,
                                       UX_HOST_CLASS_AUDIO_TRANSFER_REQUEST
                                           *audio_transfer_request)
```

Description

This function reads from the audio interface. The call is non-blocking. The application must ensure that the appropriate alternate setting has been selected for the audio streaming interface.

Parameters

audio	Pointer to the audio class instance.
audio_transfer_request	Pointer to the audio transfer structure.

Return values

UX_SUCCESS	(0x00)	The data transfer was completed
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported

Example

```
/* The following example reads from the audio interface. */

audio_transfer_request.ux_host_class_audio_transfer_request_completion_function =
                                               tx_audio_transfer_completion_function;
audio_transfer_request.ux_host_class_audio_transfer_request_class_instance = audio;
audio_transfer_request.ux_host_class_audio_transfer_request_next_audio_audio_transfer_request =
                                               UX_NULL;
audio_transfer_request. ux_host_class_audio_transfer_request_data_pointer =
                                               audio_buffer;
audio_transfer_request.ux_host_class_audio_transfer_request_requested_length =
                                               requested_length;
audio_transfer_request.ux_host_class_audio_transfer_request_packet_length =
                                               AUDIO_FRAME_LENGTH;

status = ux_host_class_audio_read(audio, audio_transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_write

Write to the audio interface

Prototype

```
UINT  ux_host_class_audio_write(UX_HOST_CLASS_AUDIO *audio,
              UX_HOST_CLASS_AUDIO_TRANSFER_REQUEST *audio_transfer_request)
```

Description

This function writes to the audio interface. The call is non-blocking. The application must ensure that the appropriate alternate setting has been selected for the audio streaming interface.

Parameters

audio	Pointer to the audio class instance
audio_transfer_request	Pointer to the audio transfer structure

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported.
UX_HOST_CLASS_AUDIO_WRONG_INTERFACE	(0x81)	Interface incorrect.

Example

```
UINT   status;

/* The following example writes to the audio interface */

audio_transfer_request.ux_host_class_audio_transfer_request_completion_function =
                                              tx_audio_transfer_completion_function;
audio_transfer_request.ux_host_class_audio_transfer_request_class_instance = audio;
audio_transfer_request.ux_host_class_audio_transfer_request_next_audio_audio_transfer_request =
                                              UX_NULL;
audio_transfer_request.ux_host_class_audio_transfer_request_data_pointer =
                                              audio_buffer;
audio_transfer_request.ux_host_class_audio_transfer_request_requested_length =
                                              requested_length;
audio_transfer_request.ux_host_class_audio_transfer_request_packet_length =
                                              AUDIO_FRAME_LENGTH;

status = ux_host_class_audio_write(audio, audio_transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_control_get

Get a specific control from the audio control interface

Prototype

```
UINT  ux_host_class_audio_control_get(UX_HOST_CLASS_AUDIO *audio,
                                      UX_HOST_CLASS_AUDIO_CONTROL
                                      *audio_control)
```

Description

This function reads a specific control from the audio control interface.

Parameters

audio	Pointer to the audio class instance
audio_control	Pointer to the audio control structure

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported
UX_HOST_CLASS_AUDIO_WRONG_INTERFACE	(0x81)	Interface incorrect

Example

```
UINT   status;

/* The following example reads the volume control from a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_CONTROL      audio_control;

audio_control. ux_host_class_audio_control_channel = 1;
audio_control. ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;

status = ux_host_class_audio_control_get(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */

audio_control. ux_host_class_audio_control_channel = 2;
audio_control. ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;

status = ux_host_class_audio_control_get(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_control_value_set

Set a specific control to the audio control interface

Prototype

```
UINT  ux_host_class_audio_control_value_set(UX_HOST_CLASS_AUDIO *audio,
                                    UX_HOST_CLASS_AUDIO_CONTROL *audio_control)
```

Description

This function sets a specific control to the audio control interface.

Parameters

audio	Pointer to the audio class instance
audio_control	Pointer to the audio control structure

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported
UX_HOST_CLASS_AUDIO_WRONG_INTERFACE	(0x81)	Interface incorrect

Example

```
/* The following example sets the volume control of a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_CONTROL   audio_control;
UINT   status;

audio_control. ux_host_class_audio_control_channel = 1;
audio_control. ux_host_class_audio_control =
UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
audio_control. ux_host_class_audio_control_cur = 0xf000;

status = ux_host_class_audio_control_value_set(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */
current_volume = audio_control.audio_ control_cur;

audio_control. ux_host_class_audio_control_channel = 2;
audio_control. ux_host_class_audio_control =
   UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
audio_control. ux_host_class_audio_control_cur = 0xf000;

status = ux_host_class_audio_control_value_set(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_streaming_sampling_set

Set an alternate setting interface of the audio streaming interface

Prototype

```
UINT  ux_host_class_audio_streaming_sampling_set(UX_HOST_CLASS_AUDIO *audio,
                                UX_HOST_CLASS_AUDIO_SAMPLING *audio_sampling)
```

Description

This function sets the appropriate alternate setting interface of the audio streaming interface according to a specific sampling structure.

Parameters

audio	Pointer to the audio class instance.
audio_sampling	Pointer to the audio sampling structure.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported
UX_HOST_CLASS_AUDIO_WRONG_INTERFACE	(0x81)	Interface incorrect
UX_NO_ALTERNATE_SETTING	(0x5e)	No alternate setting for the sampling values

Example

```
/* The following example sets the alternate setting interface of a
   stereo USB speaker. */

UX_HOST_CLASS_AUDIO_SAMPLING   audio_sampling;
UINT   status;

sampling. ux_host_class_audio_sampling_channels = 2;
sampling. ux_host_class_audio_sampling_frequency = AUDIO_FREQUENCY;
sampling. ux_host_class_audio_sampling_resolution = 16;

status = ux_host_class_audio_streaming_sampling_set(audio, &sampling);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_streaming_sampling_get

Get possible sampling settings of audio streaming interface

Prototype

```
UINT  ux_host_class_audio_streaming_sampling_get(UX_HOST_CLASS_AUDIO *audio,
            UX_HOST_CLASS_AUDIO_SAMPLING_CHARACTERISTICS *audio_sampling)
```

Description

This function gets, one by one, all the possible sampling settings available in each of the alternate settings of the audio streaming interface. The first time the function is used, all the fields in the calling structure pointer must be reset. The function will return a specific set of streaming values upon return unless the end of the alternate settings has been reached. When this function is reused, the previous sampling values will be used to find the next sampling values.

Parameters

audio	Pointer to the audio class instance
audio_sampling	Pointer to the audio sampling structure

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported
UX_HOST_CLASS_AUDIO_WRONG_INTERFACE	(0x81)	Interface incorrect
UX_NO_ALTERNATE_SETTING	(0x5e)	No alternate setting for the sampling values

Example

```
/* The following example gets the sampling values for the first alternate
   setting interface of a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_SAMPLING_CHARACTERISTICS   audio_sampling;
UINT  status;

sampling.ux_host_class_audio_sampling_channels=0;
sampling.ux_host_class_audio_sampling_frequency_low=0;
sampling.ux_host_class_audio_sampling_frequency_high=0;
sampling.ux_host_class_audio_sampling_resolution=0;

status = ux_host_class_audio_streaming_sampling_get(audio, &sampling);

/* If status equals UX_SUCCESS, the operation was successful and information
 could be displayed as follows:

  printf("Number of channels %d, Resolution %d bits, frequency range %d-%d\n",
         sampling.audio_channels, sampling.audio_resolution,
         sampling.audio_frequency_low, sampling.audio_frequency_high);
*/
```

ux_host_class_asix_read

Read from the asix interface

Prototype

```
UINT  ux_host_class_asix_read(UX_HOST_CLASS_ASIX *asix, UCHAR *data_pointer,
                              ULONG requested_length, ULONG *actual_length)
```

Description

This function reads from the asix interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

asix	Pointer to the asix class instance.
data_pointer	Pointer to the buffer address of the data payload.
requested_length	Length to be received.
actual_length	Length actually received.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, reading incomplete.

Example

```
UINT   status;

/* The following example illustrates this service. */

status = ux_host_class_asix_read(asix, data_pointer,
                                 requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_asix_write

Write to the asix interface

Prototype

```
UINT  ux_host_class_asix_write(VOID *asix_class, NX_PACKET *packet)
```

Description

This function writes to the asix interface. The call is non blocking.

Parameters

asix	Pointer to the asix class instance.
packet	Netx data packet

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_ERROR	(0xFF)	Transfer could not be requested.

Example

```
UINT   status;

/* The following example illustrates this service. */

status = ux_host_class_asix_write(asix, packet);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_pima_session_open

Open a session between Initiator and Responder

Prototype

```
UINT  ux_host_class_pima_session_open(UX_HOST_CLASS_PIMA *pima,
                                      UX_HOST_CLASS_PIMA_SESSION *pima_session)
```

Description

This function opens a session between a PIMA Initiator and a PIMA Responder. Once a session is successfully opened, most PIMA commands can be executed.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session

Return Values

UX_SUCCESS	(0x00)	Session successfully opened
UX_HOST_CLASS_PIMA_RC_SESSION_ALREADY_OPENED	(0x201E)	Session already opened

Example

```
/* Open a pima session.  */
status = ux_host_class_pima_session_open(pima, pima_session);

if (status != UX_SUCCESS)
    return(UX_PICTBRIDGE_ERROR_SESSION_NOT_OPEN);
```

ux_host_class_pima_session_close

Close a session between Initiator and Responder

Prototype

```
UINT  ux_host_class_pima_session_close(UX_HOST_CLASS_PIMA *pima,
                                       UX_HOST_CLASS_PIMA_SESSION *pima_session)
```

Description

This function closes a session that was previously opened between a PIMA Initiator and a PIMA Responder. Once a session is closed, most PIMA commands can no longer be executed.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session

Return Values

UX_SUCCESS	(0x00)	The session was closed
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened

Example

```
/* Close the pima session.  */
status = ux_host_class_pima_session_close(pima, pima_session);
```

ux_host_class_pima_storage_ids_get

Obtain the storage ID array from Responder

Prototype

```
UINT  ux_host_class_pima_storage_ids_get(UX_HOST_CLASS_PIMA *pima,
                                         UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                         ULONG *storage_ids_array,
                                         ULONG storage_id_length)
```

Description

This function obtains the storage ID array from the responder.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
storage_ids_array	Array where storage IDs will be returned
storage_id_length	Length of the storage array

Return Values

UX_SUCCESS	(0x00)	The storage ID array has been populated
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Get the number of storage IDs.  */
status = ux_host_class_pima_storage_ids_get(pima, pima_session,
                        pictbridge -> ux_pictbridge_storage_ids, 64);

if (status != UX_SUCCESS)
{

    /* Close the pima session.  */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_storage_info_get

Obtain the storage information from Responder

Prototype

```
UINT  ux_host_class_pima_storage_info_get(UX_HOST_CLASS_PIMA *pima,
                                    UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                    ULONG storage_id,
                                    UX_HOST_CLASS_PIMA_STORAGE *storage)
```

Description

This function obtains the storage information for a storage container of value storage_id

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
storage_id	ID of the storage container
storage	Pointer to storage information container

Return Values

UX_SUCCESS	(0x00)	The storage information was retrieved
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Get the first storage ID info container.   */
status = ux_host_class_pima_storage_info_get(pima, pima_session,
                        pictbridge -> ux_pictbridge_storage_ids[0],
                        (UX_HOST_CLASS_PIMA_STORAGE *)pictbridge ->
                             ux_pictbridge_storage);
if (status != UX_SUCCESS)
{
   /* Close the pima session.  */
   status = ux_host_class_pima_session_close(pictbridge ->
                                    ux_pictbridge_pima, pima_session);

   return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_num_objects_get

Obtain the number of objects on a storage container from Responder

Prototype

```
UINT  ux_host_class_pima_num_objects_get(UX_HOST_CLASS_PIMA *pima,
                                         UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                         ULONG storage_id,
                                         ULONG object_format_code)
```

Description

This function obtains the number of objects stored on a specific storage container of value storage_id matching a specific format code. The number of objects is returned in the field: ux_host_class_pima_session_nb_objects of the pima_session structure.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
storage_id	ID of the storage container
object_format_code	Objects format code filter.

The Object Format Codes can have one of the following values:

Object Format Code	Description	USBX code
0x3000	Undefined Undefined non-image object	UX_HOST_CLASS_PIMA_OFC_UNDEFINED
0x3001	Association Association (e.g. folder)	UX_HOST_CLASS_PIMA_OFC_ASSOCIATION
0x3002	Script Device-model-specific script	UX_HOST_CLASS_PIMA_OFC_SCRIPT
0x3003	Executable Device-model-specific binary executable	UX_HOST_CLASS_PIMA_OFC_EXECUTABLE
0x3004	Text Text file	UX_HOST_CLASS_PIMA_OFC_TEXT
0x3005	HTML HyperText Markup Language file (text)	UX_HOST_CLASS_PIMA_OFC_HTML
0x3006	DPOF Digital Print Order Format file (text)	UX_HOST_CLASS_PIMA_OFC_DPOF
0x3007	AIFF Audio clip	UX_HOST_CLASS_PIMA_OFC_AIFF
0x3008	WAV Audio clip	UX_HOST_CLASS_PIMA_OFC_WAV

0x3009	MP3 Audio clip	UX_HOST_CLASS_PIMA_OFC_MP3
0x300A	AVI Video clip	UX_HOST_CLASS_PIMA_OFC_AVI
0x300B	MPEG Video clip	UX_HOST_CLASS_PIMA_OFC_MPEG
0x300C	ASF Microsoft Advanced Streaming Format (video)	UX_HOST_CLASS_PIMA_OFC_ASF
0x3800	Undefined Unknown image object	UX_HOST_CLASS_PIMA_OFC_QT
0x3801	EXIF/JPEG Exchangeable File Format, JEIDA standard	UX_HOST_CLASS_PIMA_OFC_EXIF_JPEG
0x3802	TIFF/EP Tag Image File Format for Electronic Photography	UX_HOST_CLASS_PIMA_OFC_TIFF_EP
0x3803	FlashPix Structured Storage Image Format	UX_HOST_CLASS_PIMA_OFC_FLASHPIX
0x3804	BMP Microsoft Windows Bitmap file	UX_HOST_CLASS_PIMA_OFC_BMP
0x3805	CIFF Canon Camera Image File Format	UX_HOST_CLASS_PIMA_OFC_CIFF
0x3806	Undefined Reserved	
0x3807	GIF Graphics Interchange Format	UX_HOST_CLASS_PIMA_OFC_GIF
0x3808	JFIF JPEG File Interchange Format	UX_HOST_CLASS_PIMA_OFC_JFIF
0x3809	PCD PhotoCD Image Pac	UX_HOST_CLASS_PIMA_OFC_PCD
0x380A	PICT Quickdraw Image Format	UX_HOST_CLASS_PIMA_OFC_PICT
0x380B	PNG Portable Network Graphics	UX_HOST_CLASS_PIMA_OFC_PNG
0x380C	Undefined Reserved	
0x380D	TIFF Tag Image File Format	UX_HOST_CLASS_PIMA_OFC_TIFF
0x380E	TIFF/IT Tag Image File Format for Information Technology (graphic arts)	UX_HOST_CLASS_PIMA_OFC_TIFF_IT
0x380F	JP2 JPEG2000 Baseline File Format	UX_HOST_CLASS_PIMA_OFC_JP2
0x3810	JPX JPEG2000 Extended File Format	UX_HOST_CLASS_PIMA_OFC_JPX

All other codes with MSN of 0011	Any Undefined Reserved for future use	
All other codes with MSN of 1011	Any Vendor-Defined Vendor-Defined type*:* Image	

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Get the number of objects on all containers matching a SCRIPT
      object.  */
status = ux_host_class_pima_num_objects_get(pima, pima_session,
            UX_PICTBRIDGE_ALL_CONTAINERS, UX_PICTBRIDGE_OBJECT_SCRIPT);
if (status != UX_SUCCESS)
{
      /* Close the pima session.  */
     status = ux_host_class_pima_session_close(pima, pima_session);

     return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
else
      /* The number of objects is returned in the field: pima_session -
      > ux_host_class_pima_session_nb_objects */
```

ux_host_class_pima_object_handles_get

Obtain object handles from Responder

Prototype

```
UINT  ux_host_class_pima_object_handles_get(UX_HOST_CLASS_PIMA *pima,
                                    UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                    ULONG *object_handles_array,
                                    ULONG object_handles_length,
                                    ULONG storage_id,
                                    ULONG object_format_code,
                                    ULONG object_handle_association)
```

Description

Returns an array of Object Handles present in the storage container indicated by the storage_id parameter. If an aggregated list across all stores is desired, this value shall be set to 0xFFFFFFFF.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handes_array	Array where handles are returned
object_handles_length	Length of the array
storage_id	ID of the storage container
object_format_code	Format code for object (see table for function ux_host_class_pima_num_objects_get)
object_handle_association	Optional object association value

The object handle association can be one of the value from the table below:

AssociationCode	AssociationType	AssociationDesc Interpretation
0x0000	Undefined	Undefined
0x0001	GenericFolder	Unused
0x0002	Album	Reserved
0x0003	TimeSequence	DefaultPlaybackDelta
0x0004	HorizontalPanoramic	Unused
0x0005	VerticalPanoramic	Unused
0x0006	2DPanoramic	ImagesPerRow
0x0007	AncillaryData	Undefined
All other values with bit 15 set to 0	Reserved	Undefined
All values with bit 15 set to 1	Vendor-Defined	Vendor-Defined

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Get the array of objects handles on the container.  */
status = ux_host_class_pima_object_handles_get(pima, pima_session,
             pictbridge -> ux_pictbridge_object_handles_array,
             4 * pima_session -> ux_host_class_pima_session_nb_objects,
             UX_PICTBRIDGE_ALL_CONTAINERS,
             UX_PICTBRIDGE_OBJECT_SCRIPT, 0);

if (status != UX_SUCCESS)
{
    /* Close the pima session.  */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_object_info_get

Obtain the object information from Responder

Prototype

```
UINT  ux_host_class_pima_object_info_get(UX_HOST_CLASS_PIMA *pima,
                                         UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                         ULONG object_handle,
                                         UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function obtains the object information for an object handle.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	Handle of the object
object	Pointer to object information container

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* We search for an object that is a picture or a script.  */
object_index =  0;
while (object_index < pima_session ->
                                    ux_host_class_pima_session_nb_objects)
{

    /* Get the object info structure.  */
    status = ux_host_class_pima_object_info_get(pima, pima_session,
                      pictbridge ->
                        ux_pictbridge_object_handles_array[object_index],
                      pima_object);
    if (status != UX_SUCCESS)
    {
        /* Close the pima session.  */
        status = ux_host_class_pima_session_close(pima, pima_session);

        return(UX_PICTBRIDGE_ERROR_INVALID_OBJECT_HANDLE );
    }
}
```

ux_host_class_pima_object_info_send

Send the object information to Responder

Prototype

```
UINT  ux_host_class_pima_object_info_send(UX_HOST_CLASS_PIMA *pima,
                                          UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                          ULONG storage_id,
                                          ULONG parent_object_id,
                                          UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function sends the storage information for a storage container of value storage_id. The Initiator should use this command before sending an object to the responder.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
storage_id	Destination storage ID
parent_object_id	Parent ObjectHandle on Responder where object should be placed
object	Pointer to object information container

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Send a script info.  */
status = ux_host_class_pima_object_info_send(pima, pima_session,
                                             0, 0, pima_object);
if (status != UX_SUCCESS)
{
    /* Close the pima session.  */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_ERROR );
}
```

ux_host_class_pima_object_open

Open an object stored in the Responder

Prototype

```
UINT  ux_host_class_pima_object_open(UX_HOST_CLASS_PIMA *pima,
                                     UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                     ULONG object_handle,
                                     UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function opens an object on the responder before reading or writing.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	handle of the object
object	Pointer to object information container

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_HOST_CLASS_PIMA_RC_OBJECT_ALREADY_OPENED	(0x2021)	Object already opened.
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Open the object.  */
status = ux_host_class_pima_object_open(pima, pima_session,
                                        object_handle, pima_object);

/* Check status.  */
if (status != UX_SUCCESS)
   return(status);
```

ux_host_class_pima_object_get

Get an object stored in the Responder

Prototype

```
UINT  ux_host_class_pima_object_get(UX_HOST_CLASS_PIMA *pima,
                                    UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                    ULONG object_handle,
                                    UX_HOST_CLASS_PIMA_OBJECT *object,
                                    UCHAR *object_buffer,
                                    ULONG object_buffer_length,
                                    ULONG *object_actual_length)
```

Description

This function gets an object on the responder.

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	handle of the object
object	Pointer to object information container
object_buffer	Address of object data
object_buffer_length	Requested length of object
object_actual_length	Length of object returned

Return Values

UX_SUCCESS	(0x00)	The object was transfered
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED	(0x2023)	Object not opened.
UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED	(0x200f)	Access to object denied
UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER	(0x2007)	Transfer is incomplete
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.
UX_TRANSFER_ERROR	(0x23)	Transfer error while reading object

Example

```
/* Open the object.  */
status = ux_host_class_pima_object_open(pima, pima_session,
                                        object_handle, pima_object);
/* Check status.  */
if (status != UX_SUCCESS)
    return(status);

/* Set the object buffer pointer.  */
object_buffer = pima_object -> ux_host_class_pima_object_buffer;

/* Obtain all the object data.  */
while(object_length != 0)
{
    /* Calculate what length to request.  */
    if (object_length > UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER)

        /* Request maximum length.  */
        requested_length = UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER;
    else

        /* Request remaining length.  */
        requested_length = object_length;

    /* Get the object data.  */
    status = ux_host_class_pima_object_get(pima, pima_session,
                            object_handle, pima_object, object_buffer,
                            requested_length, &actual_length);

    if (status != UX_SUCCESS)
    {

        /* We had a problem, abort the transfer.  */
        ux_host_class_pima_object_transfer_abort(pima, pima_session,
                                          object_handle, pima_object);

        /* And close the object.  */
        ux_host_class_pima_object_close(pima, pima_session,
                                        object_handle, pima_object,
                                        object);

        return(status);
    }

    /* We have received some data, update the length remaining.  */
    object_length -= actual_length;

    /* Update the buffer address.  */
    object_buffer += actual_length;
}

/* Close the object.  */
status = ux_host_class_pima_object_close(pima, pima_session,
                                         object_handle, pima_object,
                                         object);
```

ux_host_class_pima_object_send

Send an object stored in the Responder

Prototype

```
UINT  ux_host_class_pima_object_send(UX_HOST_CLASS_PIMA *pima,
                            UX_HOST_CLASS_PIMA_SESSION *pima_session,
                            UX_HOST_CLASS_PIMA_OBJECT *object,
                            UCHAR *object_buffer, ULONG object_buffer_length)
```

Description

This function sends an object to the responder

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	handle of the object
object	Pointer to object information container
object_buffer	Address of object data
object_buffer_length	Requested length of object

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED	(0x2023)	Object not opened.
UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED	(0x200f)	Access to object denied
UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER	(0x2007)	Transfer is incomplete
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.
UX_TRANSFER_ERROR	(0x23)	Transfer error while writing object

Example

```
/* Open the object.  */
status = ux_host_class_pima_object_open(pima, pima_session,
                                        object_handle,
                                        pima_object);
/* Get the object length.  */
object_length =  pima_object -> ux_host_class_pima_object_compressed_size;

/* Recall the object buffer address.  */
pima_object_buffer = pima_object -> ux_host_class_pima_object_buffer;

/* Send all the object data.  */
while(object_length != 0)
{

    /* Calculate what length to request.  */
    if (object_length > UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER)

        /* Request maximum length.  */
        requested_length = UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER;

    else

        /* Request remaining length.  */
        requested_length = object_length;


    /* Send the object data.  */
    status = ux_host_class_pima_object_send(pima, pima_session, pima_object,
                                        pima_object_buffer, requested_length);
    if (status != UX_SUCCESS)
    {

        /* Abort the transfer.  */
        ux_host_class_pima_object_transfer_abort(pima, pima_session,
                                                 object_handle, pima_object);

        /* Return status.  */
        return(status);

    }

    /* We have sent some data, update the length remaining.  */
    object_length -= requested_length;
}

/* Close the object.  */
status = ux_host_class_pima_object_close(pima, pima_session, object_handle,
                                         pima_object, object);
```

ux_host_class_pima_thumb_get

Get a thumb object stored in the Responder

Prototype

```
UINT  ux_host_class_pima_thumb_get(UX_HOST_CLASS_PIMA *pima,
                UX_HOST_CLASS_PIMA_SESSION *pima_session,
                ULONG object_handle, UX_HOST_CLASS_PIMA_OBJECT *object,
                UCHAR *thumb_buffer, ULONG thumb_buffer_length,
                ULONG *thumb_actual_length)
```

Description

This function gets a thumb object on the responder

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	handle of the object
object	Pointer to object information container
thumb_buffer	Address of thumb object data
thumb_buffer_length	Requested length of thumb object
thumb_actual_length	Length of thumb object returned

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED	(0x2023)	Object not opened.
UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED	(0x200f)	Access to object denied
UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER	(0x2007)	Transfer is incomplete
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.
UX_TRANSFER_ERROR	(0x23)	Transfer error while reading object

Example

```
/* Get the thumb object data.  */
status = ux_host_class_pima_thumb_get(pima, pima_session,
                            object_handle, pima_object, object_buffer,
                            requested_length, &actual_length);

if (status != UX_SUCCESS)
{

   /* And close the object.  */
   ux_host_class_pima_object_close(pima, pima_session,
                               object_handle, pima_object, object);

   return(status);
}
```

ux_host_class_pima_object_delete

Delete an object stored in the Responder

Prototype

```
UINT  ux_host_class_pima_object_delete(UX_HOST_CLASS_PIMA *pima,
                                       UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                       ULONG object_handle)
```

Description

This function deletes an object on the responder

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	handle of the object

Return Values

UX_SUCCESS	(0x00)	The object was deleted.
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED	(0x200f)	Cannot delete object
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Delete the object.  */
status = ux_host_class_pima_object_delete(pima, pima_session,
                                          object_handle, pima_object);

/* Check status.  */
if (status != UX_SUCCESS)
   return(status);
```

ux_host_class_pima_object_close

Close an object stored in the Responder

Prototype

```
UINT  ux_host_class_pima_object_close(UX_HOST_CLASS_PIMA *pima,
                                      UX_HOST_CLASS_PIMA_SESSION *pima_session,
                                      ULONG object_handle,
                                      UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function closes an object on the responder

Parameters

pima	Pointer to the pima class instance.
pima_session	Pointer to PIMA session
object_handle	Pandle of the object
object	Pointer to object

Return Values

UX_SUCCESS	(0x00)	The object was closed
UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN	(0x2003)	Session not opened
UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED	(0x2023)	Object not opened.
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to create PIMA command.

Example

```
/* Close the object.  */
status = ux_host_class_pima_object_close(pima, pima_session,
                                         object_handle, object);
```

ux_host_class_gser_read

Read from the generic serial interface

Prototype

```
UINT  ux_host_class_gser_read(UX_HOST_CLASS_GSER *gser,
                              ULONG interface_index, UCHAR *data_pointer,
                              ULONG requested_length,
                              ULONG *actual_length)
```

Description

This function reads from the generic serial interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

gser	Pointer to the gser class instance.
interface_index	Interface index to read from
data_pointer	Pointer to the buffer address of the data payload
requested_length	Length to be received.
actual_length	Length actually received.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, reading incomplete.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_gser_read(cdc_acm, interface_index,data_pointer,
                                 requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_write

Write to the generic serial interface

Prototype

```
UINT  ux_host_class_gser_write(UX_HOST_CLASS_GSER *gser,
                               ULONG interface_index, UCHAR *data_pointer,
                               ULONG requested_length, ULONG *actual_length)
```

Description

This function writes to the generic serial interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

gser	Pointer to the gser class instance.
interface_index	Interface to which to write
data_pointer	Pointer to the buffer address of the data payload.
requested_length	Length to be sent.
actual_length	Length actually sent.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, writing incomplete.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_cdc_acm_write(gser, data_pointer,
                                     requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_ioctl

Perform an IOCTL function to the generic serial interface

Prototype

```
UINT  ux_host_class_gser_ioctl(UX_HOST_CLASS_GSER *gser,
                               ULONG ioctl_function,
                               VOID *parameter)
```

Description

This function performs a specific ioctl function to the gser interface. The call is blocking and only returns when there is either an error or when the command is completed.

Parameters

gser	Pointer to the gser class instance.
ioctl_function	ioctl function to be performed. See table below for one of the allowed ioctl functions.
parameter	Pointerto a parameter specific to the ioctl

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory.
UX_HOST_CLASS_UNKNOWN	(0x59)	Wrong class instance
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Unknown IOCTL function

IOCTL functions:

UX_HOST_CLASS_GSER_IOCTL_SET_LINE_CODING
UX_HOST_CLASS_GSER_IOCTL_GET_LINE_CODING
UX_HOST_CLASS_GSER_IOCTL_SET_LINE_STATE
UX_HOST_CLASS_GSER_IOCTL_SEND_BREAK
UX_HOST_CLASS_GSER_IOCTL_ABORT_IN_PIPE
UX_HOST_CLASS_GSER_IOCTL_ABORT_OUT_PIPE
UX_HOST_CLASS_GSER_IOCTL_NOTIFICATION_CALLBACK
UX_HOST_CLASS_GSER_IOCTL_GET_DEVICE_STATUS

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_gser_ioctl(gser,
                                  UX_HOST_CLASS_GSER_IOCTL_GET_LINE_CODING,
                                  (VOID *)&line_coding);
/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_reception_start

Start reception on the generic serial interface

Prototype

```
UINT  ux_host_class_gser_reception_start(UX_HOST_CLASS_GSER *gser,
                                UX_HOST_CLASS_GSER_RECEPTION *gser_reception)
```

Description

This function starts the reception on the generic serial class interface. This function allows for non-blocking reception. When a buffer is received, a callback in invoked into the application.

Parameters

gser	Pointer to the gser class instance.
gser_reception	Structure containing the reception parameters

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_UNKNOWN	(0x59)	Wrong class instance
UX_ERROR	(0x01)	Error

Example

```
    /* Start the reception for gser.  AT commands are on interface 2. */
gser_reception.ux_host_class_gser_reception_interface_index =
                                        UX_DEMO_GSER_AT_INTERFACE;
gser_reception.ux_host_class_gser_reception_block_size =
                                        UX_DEMO_RECEPTION_BLOCK_SIZE;
gser_reception.ux_host_class_gser_reception_data_buffer =
                                        gser_reception_buffer;
gser_reception.ux_host_class_gser_reception_data_buffer_size =
                                        UX_DEMO_RECEPTION_BUFFER_SIZE;
gser_reception.ux_host_class_gser_reception_callback =
                                        tx_demo_thread_callback;
ux_host_class_gser_reception_start(gser, &gser_reception);
```

ux_host_class_gser_reception_stop

Stop reception on the generic serial interface

Prototype

```
UINT  ux_host_class_gser_reception_stop(UX_HOST_CLASS_GSER *gser,
                                        UX_HOST_CLASS_GSER_RECEPTION *gser_reception)
```

Description

This function stops the reception on the generic serial class interface.

Parameters

gser	Pointer to the gser class instance.
gser_reception	Structure containing the reception parameters

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_HOST_CLASS_UNKNOWN	(0x59)	Wrong class instance
UX_ERROR	(0x01)	Error

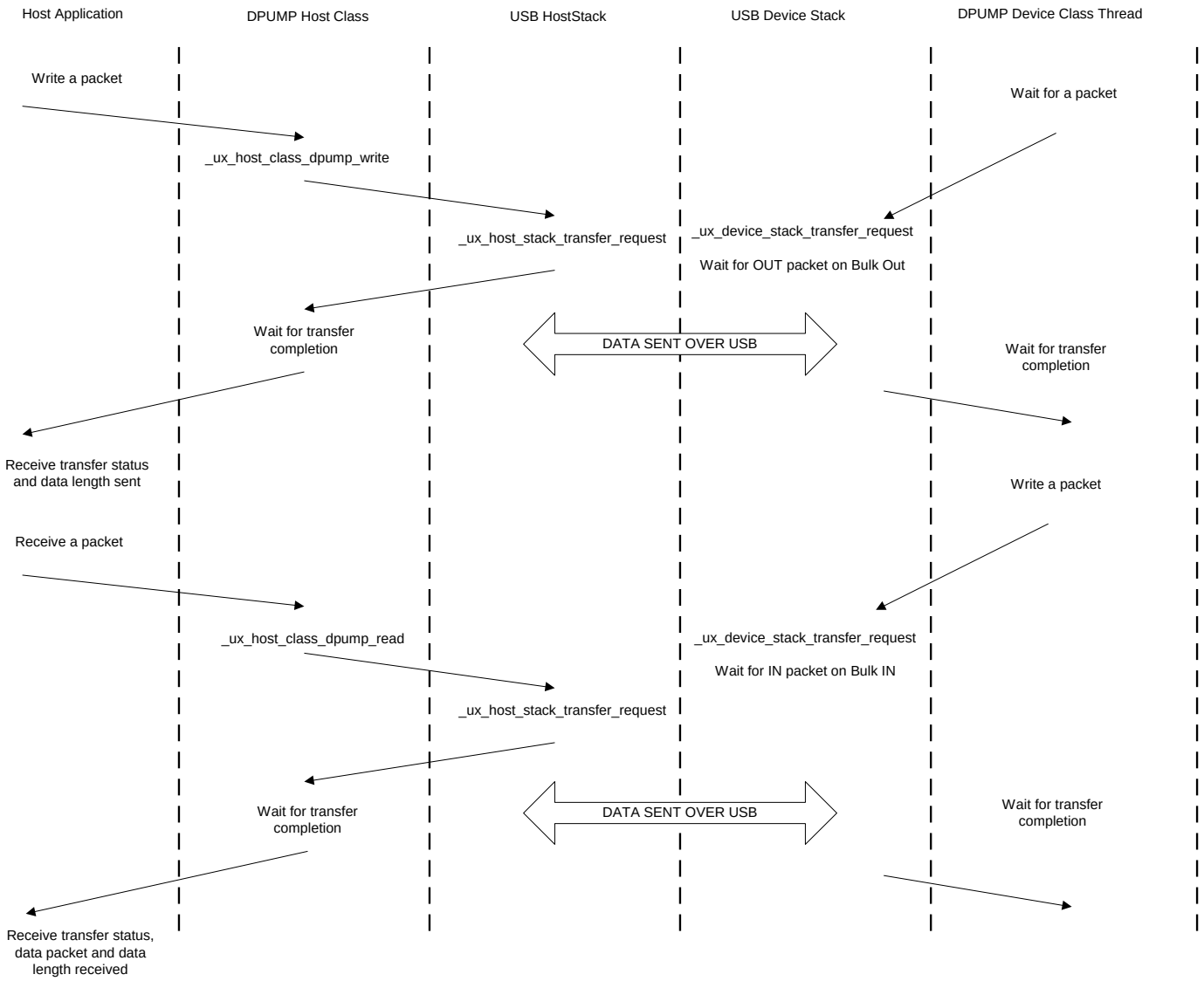
Example

```
/* Stops the reception for gser. */
ux_host_class_gser_reception_stop(gser, &gser_reception);
```

Chapter 3: USBX DPUMP Class Considerations

USBX contains a DPUMP class for the host and device side. This class is not a standard class per se, but rather an example that illustrates how to create a simple device by using 2 bulk pipes and sending data back and forth on these 2 pipes. The DPUMP class could be used to start a custom class or for legacy RS232 devices.

USB DPUMP flow chart:



USBX DPUMP Host Class

The host side of the DPUMP Class has 2 functions, one for sending data and one for receiving data:

```
ux_host_class_dpump_write
ux_host_class_dpump_read
```

Both functions are blocking to make the DPUMP application easier. If it is necessary to have both pipes (IN and OUT) running at the same time, the application will be required to create a transmit thread and a receive thread.

The prototype for the writing function is as follows:

```
UINT  ux_host_class_dpump_write(UX_HOST_CLASS_DPUMP *dpump,
                                UCHAR * data_pointer,
                                ULONG requested_length,
                                ULONG *actual_length)
```

Where:

dpump is the instance of the class
data_pointer is the pointer to the buffer to be sent
requested_length is the length to send
actual_length is the length sent after completion of the transfer, either successfully or partially.

The prototype for the receiving function is the same:

```
UINT  ux_host_class_dpump_read(UX_HOST_CLASS_DPUMP *dpump,
                               UCHAR *data_pointer,
                               ULONG requested_length,
                               ULONG *actual_length)
```

Here is an example of the host DPUMP class where an application writes a packet to the device side and receives the same packet on the reception:

```
/* We start with a 'A' in buffer.  */
current_char = 'A';

while(1)
{
    /* Initialize the write buffer. */
    ux_utility_memory_set(out_buffer, current_char,
                          UX_HOST_CLASS_DPUMP_PACKET_SIZE);

    /* Increment the character in buffer.  */
    current_char++;


    /* Check for upper alphabet limit.  */
    if (current_char > 'Z')
       current_char =  'A';
```

```
/* Write to the Data Pump Bulk out endpoint.  */
status =  ux_host_class_dpump_write (dpump, out_buffer,
                                     UX_HOST_CLASS_DPUMP_PACKET_SIZE,
                                     &actual_length);

/* Verify that the status and the amount of data is correct.  */
if ((status == UX_SUCCESS) && actual_length ==
                                     UX_HOST_CLASS_DPUMP_PACKET_SIZE)

{
    /* Read to the Data Pump Bulk out endpoint.  */
    status =  ux_host_class_dpump_read (dpump, in_buffer,
                      UX_HOST_CLASS_DPUMP_PACKET_SIZE, &actual_length);

}
```

USBX DPUMP Device Class

The device DPUMP class uses a thread which is started upon connection to the USB host. The thread waits for a packet coming on the Bulk Out endpoint. When a packet is received, it copies the content to the Bulk In endpoint buffer and posts a transaction on this endpoint, waiting for the host to issue a request to read from this endpoint. This provides a loopback mechanism between the Bulk Out and Bulk In endpoints.

Chapter 4: USBX Pictbridge implementation

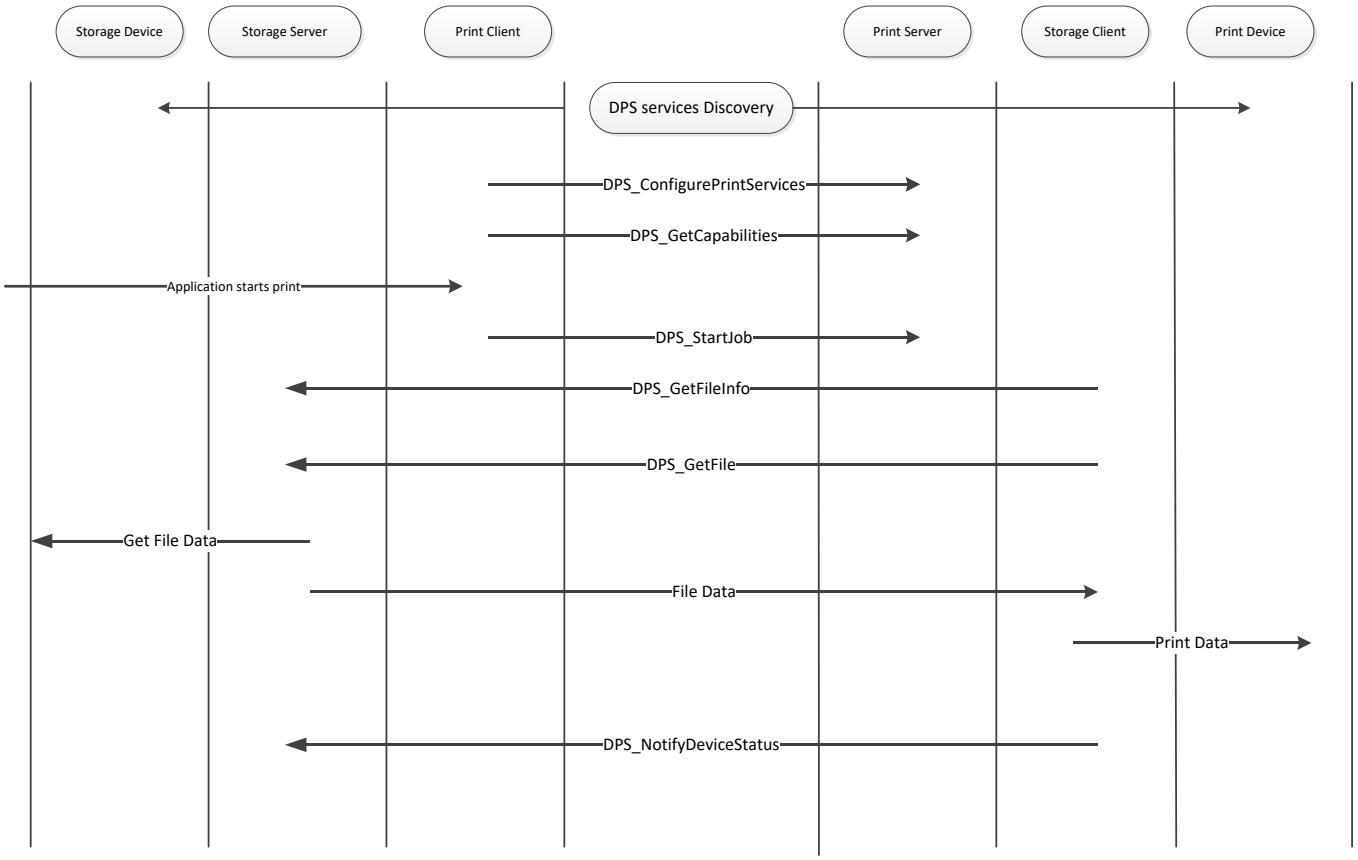
UBSX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of USBX PIMA class on both sides.

The PictBridge standards allows the connection of a digital still camera or a smart phone directly to a printer without a PC, enabling direct printing to certain Pictbridge aware printers.

When a camera or phone is connected to a printer, the printer is the USB host and the camera is the USB device. However, with Pictbridge, the camera will appear as being the host and commands are driven from the camera. The camera is the storage server, the printer the storage client. The camera is the print client and the printer is, of course, the print server.

Pictbridge uses USB as a transport layer but relies on PTP (Picture Transfer Protocol) for the communication protocol.

The following is a diagram of the commands/responses between the DPS client and the DPS server when a print job occurs:



Pictbridge client implementation

The Pictbridge on the client requires the USBX device stack and the PIMA class to be running first.

A device framework describes the PIMA class in the following way:

```
UCHAR device_framework_full_speed[] =
{

    /* Device descriptor */
        0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x20,
        0xA9, 0x04, 0xB6, 0x30, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x01,

    /* Configuration descriptor */
        0x09, 0x02, 0x27, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

    /* Interface descriptor */
        0x09, 0x04, 0x00, 0x00, 0x03, 0x06, 0x01, 0x01, 0x00,

    /* Endpoint descriptor (Bulk Out) */
        0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint descriptor (Bulk In) */
        0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint descriptor (Interrupt) */
        0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x60

};
```

The Pima class is using the ID field 0x06 and has its subclass is 0x01 for Still Image and the protocol is 0x01 for PIMA 15740.

3 endpoints are defined in this class, 2 bulks for sending/receiving data and one interrupt for events.

Unlike other USBX device implementations, the Pictbridge application does not need to define a class itself. Rather it invokes the function ux_pictbridge_dpsclient_start. An example is below:

```
    /* Initialize the Pictbridge string components. */
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name,
       "ExpressLogic",13);
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name,
       "EL_Pictbridge_Camera",21);
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no,
       "ABC_123",7);
```

```
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions,
       "1.0 1.1",7);
    pictbridge.ux_pictbridge_dpslocal.
       ux_pictbridge_devinfo_vendor_specific_version = 0x0100;
    /* Start the Pictbridge client.  */
    status = ux_pictbridge_dpsclient_start(&pictbridge);

    if(status != UX_SUCCESS)
       return;
```

The parameters passed to the pictbridge client are as follows:

```
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name
      : String of Vendor name
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name
      : String of product name
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no,
      : String of serial number
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions
      : String of version
pictbridge.ux_pictbridge_dpslocal.
      ux_pictbridge_devinfo_vendor_specific_version
      : Value set to  0x0100;
```

The next step is for the device and the host to synchronize and be ready to exchange information.

This is done by waiting on an event flag as follows:

```
/* We should wait for the host and the client to discover one another.  */
status =  ux_utility_event_flags_get
      (&pictbridge.ux_pictbridge_event_flags_group,
       UX_PICTBRIDGE_EVENT_FLAG_DISCOVERY,TX_AND_CLEAR, &actual_flags,
       UX_PICTBRIDGE_EVENT_TIMEOUT);
```

If the state machine is in the DISCOVERY_COMPLETE state, the camera side (the DPS client) will gather information regarding the printer and its capabilities.

If the DPS client is ready to accept a print job, its status will be set to UX_PICTBRIDGE_NEW_JOB_TRUE. It can be checked below:

```
/* Check if the printer is ready for a print job.  */
if (pictbridge.ux_pictbridge_dpsclient.ux_pictbridge_devinfo_newjobok ==
                  UX_PICTBRIDGE_NEW_JOB_TRUE)
      /* We can print something … */
```

Next some print joib descriptors need to be filled as follows:

```
/* We can start a new job. Fill in the JobConfig and PrintInfo structures. */
jobinfo = &pictbridge.ux_pictbridge_jobinfo;

/* Attach a printinfo structure to the job.  */
```

```
jobinfo -> ux_pictbridge_jobinfo_printinfo_start = &printinfo;


/* Set the default values for print job.  */
jobinfo -> ux_pictbridge_jobinfo_quality  =
                                  UX_PICTBRIDGE_QUALITIES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_papersize  =
                                 UX_PICTBRIDGE_PAPER_SIZES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_papertype  =
                                 UX_PICTBRIDGE_PAPER_TYPES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_filetype  =
                                  UX_PICTBRIDGE_FILE_TYPES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_dateprint  =
                                 UX_PICTBRIDGE_DATE_PRINTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_filenameprint  =
                            UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_imageoptimize  =
                                 UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF;
jobinfo -> ux_pictbridge_jobinfo_layout  =
                                    UX_PICTBRIDGE_LAYOUTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_fixedsize  =
                                  UX_PICTBRIDGE_FIXED_SIZE_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_cropping  =
                                   UX_PICTBRIDGE_CROPPINGS_DEFAULT;

/* Program the callback function for reading the object data.  */
jobinfo -> ux_pictbridge_jobinfo_object_data_read  =
                                                 ux_demo_object_data_copy;

/* This is a demo, the fileID is hardwired (1 and 2 for scripts, 3 for photo
   to be printed.  */
printinfo.ux_pictbridge_printinfo_fileid  =
                                    UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_filename,
                       "Pictbridge demo file", 20);
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_date, "01/01/2008",
                       10);

/* Fill in the object info to be printed.  First get the pointer to the
   object container in the job info structure. */
object = (UX_SLAVE_CLASS_PIMA_OBJECT *) jobinfo ->
                                          ux_pictbridge_jobinfo_object;

/* Store the object format: JPEG picture.  */
object -> ux_device_class_pima_object_format  =
      UX_DEVICE_CLASS_PIMA_OFC_EXIF_JPEG;
object -> ux_device_class_pima_object_compressed_size   =  IMAGE_LEN;
object -> ux_device_class_pima_object_offset          =  0;
object -> ux_device_class_pima_object_handle_id       =
                                          UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
object -> ux_device_class_pima_object_length          =  IMAGE_LEN;

/* File name is in Unicode.  */
ux_utility_string_to_unicode("JPEG Image", object ->
                                    ux_device_class_pima_object_filename);

/* And start the job.  */
```

```
status =ux_pictbridge_dpsclient_api_start_job(&pictbridge);
```

The Pictbridge client now has a print job to do and will fetch the image blocks at a time from the application through the callback defined in the field

```
jobinfo -> ux_pictbridge_jobinfo_object_data_read
```

The prototype of that function is defined as:

ux_pictbridge_jobinfo_object_data_read

Copying a block of data from user space for printing

Prototype

```
UINT ux_pictbridge_jobinfo_object_data_read(UX_PICTBRIDGE *pictbridge,
      UCHAR *object_buffer, ULONG object_offset, ULONG object_length,
      ULONG *actual_length)
```

Description

This function is called when the DPS client needs to retrieve a data block to print to the target Pictbridge printer.

Parameters

pictbridge	Pointer to the pictbridge class instance.
object_buffer	Pointer to object buffer
object_offset	Where we are starting to read the data block
object_length	Length to be returned
actual_length	Actual length returned

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0x01)	The application could not retrieve data.

Example

```
/* Copy the object data.  */
UINT ux_demo_object_data_copy(UX_PICTBRIDGE *pictbridge,UCHAR *object_buffer,
      ULONG object_offset, ULONG object_length, ULONG *actual_length)
{

    /* Copy the demanded object data portion.  */
    ux_utility_memory_copy(object_buffer, image  + object_offset,
                           object_length);

    /* Update the actual length.  */
    *actual_length = object_length;

    /* We have copied the requested data. Return OK.  */
    return(UX_SUCCESS);

}
```

Pictbridge host implementation

The host implementation of Pictbridge is different from the client.

The first thing to do in a Pictbridge host environment is to register the Pima class as the example below shows:

```
status =  ux_host_stack_class_register(ux_system_host_class_pima_name,
                                       ux_host_class_pima_entry);
if(status != UX_SUCCESS)
       return;
```

This class is the generic PTP layer sitting between the USB host stack and the Pictbridge layer.

The next step is to initialize the Pictbridge default values for print services as follows:

Pictbridge field	Value
DpsVersion[0] DpsVersion[1] DpsVersion[2] VendorSpecificVersion	0x00010000 0x00010001 0x00000000 0x00010000
PrintServiceAvailable	0x30010000
Qualities[0] Qualities[1] Qualities[2] Qualities[3]	UX_PICTBRIDGE_QUALITIES_DEFAULT UX_PICTBRIDGE_QUALITIES_NORMAL UX_PICTBRIDGE_QUALITIES_DRAFT UX_PICTBRIDGE_QUALITIES_FINE
PaperSizes[0] PaperSizes[1] PaperSizes[2] PaperSizes[3] PaperSizes[4]	UX_PICTBRIDGE_PAPER_SIZES_DEFAULT UX_PICTBRIDGE_PAPER_SIZES_4IX6I UX_PICTBRIDGE_PAPER_SIZES_L UX_PICTBRIDGE_PAPER_SIZES_2L UX_PICTBRIDGE_PAPER_SIZES_LETTER
PaperTypes[0] PaperTypes[1] PaperTypes[2]	UX_PICTBRIDGE_PAPER_TYPES_DEFAULT UX_PICTBRIDGE_PAPER_TYPES_PLAIN UX_PICTBRIDGE_PAPER_TYPES_PHOTO
FileTypes[0] FileTypes[1] FileTypes[2] FileTypes[3]	UX_PICTBRIDGE_FILE_TYPES_DEFAULT UX_PICTBRIDGE_FILE_TYPES_EXIF_JPEG UX_PICTBRIDGE_FILE_TYPES_JFIF UX_PICTBRIDGE_FILE_TYPES_DPOF
DatePrints[0] DatePrints[1] DatePrints[2]	UX_PICTBRIDGE_DATE_PRINTS_DEFAULT UX_PICTBRIDGE_DATE_PRINTS_OFF UX_PICTBRIDGE_DATE_PRINTS_ON
FileNamePrints[0] FileNamePrints[1] FileNamePrints[2]	UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT UX_PICTBRIDGE_FILE_NAME_PRINTS_OFF UX_PICTBRIDGE_FILE_NAME_PRINTS_ON
ImageOptimizes[0]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_DEFAULT

ImageOptimizes[1] ImageOptimizes[2]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF UX_PICTBRIDGE_IMAGE_OPTIMIZES_ON
Layouts[0] Layouts[1] Layouts[2] Layouts[3]	UX_PICTBRIDGE_LAYOUTS_DEFAULT UX_PICTBRIDGE_LAYOUTS_1_UP_BORDER UX_PICTBRIDGE_LAYOUTS_INDEX_PRINT UX_PICTBRIDGE_LAYOUTS_1_UP_BORDERLESS
FixedSizes[0] FixedSizes[1] FixedSizes[2] FixedSizes[3] FixedSizes[4] FixedSizes[5] FixedSizes[6]	UX_PICTBRIDGE_FIXED_SIZE_DEFAULT UX_PICTBRIDGE_FIXED_SIZE_35IX5I UX_PICTBRIDGE_FIXED_SIZE_4IX6I UX_PICTBRIDGE_FIXED_SIZE_5IX7I UX_PICTBRIDGE_FIXED_SIZE_7CMX10CM UX_PICTBRIDGE_FIXED_SIZE_LETTER UX_PICTBRIDGE_FIXED_SIZE_A4
Croppings[0] Croppings[1] Croppings[2]	UX_PICTBRIDGE_CROPPINGS_DEFAULT UX_PICTBRIDGE_CROPPINGS_OFF UX_PICTBRIDGE_CROPPINGS_ON

The state machine of the DPS host will be set to Idle and ready to accept a new print job.
The host portion of Pictbridge can now be started as the example below shows:

```
/* Activate the pictbridge dpshost.  */
status = ux_pictbridge_dpshost_start(&pictbridge, pima);

if (status != UX_SUCCESS)
    return;
```

The Pictbridge host function requires a callback when data is ready to be printed. This is accomplished by passing a function pointer in the pictbridge host structure as follows:

```
/* Set a callback when an object is being received.  */
pictbridge.ux_pictbridge_application_object_data_write =
                                        tx_demo_object_data_write;
```

This function has the following properties:

ux_pictbridge_application_object_data_write

Writing a block of data for printing

Prototype

```
UINT  ux_pictbridge_application_object_data_write(UX_PICTBRIDGE
            *pictbridge,UCHAR *object_buffer, ULONG offset,
            ULONG total_length, ULONG length);
```

Description

This function is called when the DPS server needs to retrieve a data block from the DPS client to print to the local printer.

Parameters

pictbridge	Pointer to the pictbridge class instance.
object_buffer	Pointer to object buffer
object_offset	Where we are starting to read the data block
total_length	Entire length of object
length	Length of this buffer

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0x01)	The application could not print data.

Example

```
/* Copy the object data.  */
UINT tx_demo_object_data_write(UX_PICTBRIDGE *pictbridge,
      UCHAR *object_buffer, ULONG offset, ULONG total_length, ULONG length);
{
UINT status;

    /* Send the data to the local printer. */
    status = local_printer_data_send(object_buffer, length);

    /* We have printed the requested data. Return status.  */
    return(status);

}
```

Chapter 5: USBX OTG

USBX supports the OTG functionalities of USB when an OTG compliant USB controller is available in the hardware design.

USBX supports OTG in the core USB stack. But for OTG to function, it requires a specific USB controller. USBX OTG controller functions can be found in the usbx_otg directory. The current USBX version only supports the NXP LPC3131 with full OTG capabilities.

The regular controller driver functions (host or device) can still be found in the standard USBX usbx_device_controllers and usbx_host_controllers but the usbx_otg directory contains the specific OTG functions associated with the USB controller.

There are 4 categories of functions for an OTG controller in addition to the usual host/device functions:

- VBUS specific functions
- Start and Stop of the controller
- USB role manager
- Interrupt handlers

VBUS functions
Each controller needs to have a VBUS manager to change the state of VBUS based on power management requirements. Usually this function only performs turning on or off VBUS

Start and Stop the controller
Unlike a regular USB implementation, OTG requires the host and/or the device stack to be activated and deactivated when the role changes.

USB role Manager
The USB role manager receives commands to change the state of the USB. There are several states that need transitions to and from:

State	Value	Description
UX_OTG_IDLE	0	The device is Idle. Usually not connected to anything
UX_OTG_IDLE_TO_HOST	1	Device is connected with type A connector
UX_OTG_IDLE_TO_SLAVE	2	Device is connected with type B connector
UX_OTG_HOST_TO_IDLE	3	Host device got disconnected
UX_OTG_HOST_TO_SLAVE	4	Role swap from Host to Slave
UX_OTG_SLAVE_TO_IDLE	5	Slave device is disconnected
UX_OTG_SLAVE_TO_HOST	6	Role swap from Slave to Host

Interrupt handlers
Both host and device controller drivers for OTG needs different interrupt handlers to monitor signals beyond traditional USB interrupts, in particular signals due to SRP and VBUS.

How to initialize a USB OTG controller. We use the NXP LPC3131 as an example here:

```
/* Initialize the LPC3131 OTG controller.  */
 status = ux_otg_lpc3131_initialize(0x19000000, lpc3131_vbus_function,
                                    tx_demo_change_mode_callback);
```

In this example, we initialize the LPC3131 in OTG mode by passing a VBUS function and a callback for mode change (from host to slave or vice versa).

The callback function should simply record the new mode and wake up a pending thread to act up the new state:

```
void  tx_demo_change_mode_callback(ULONG mode)
{

    /* Simply save the otg mode.  */
    otg_mode = mode;
```

```
    /* Wake up the thread that is waiting.  */
    ux_utility_semaphore_put(&mode_change_semaphore);

}
```

The mode value that is passed can have the following values:

- UX_OTG_MODE_IDLE
- UX_OTG_MODE_SLAVE
- UX_OTG_MODE_HOST

The application can always check what the device is by looking at the variable:

```
ux_system_otg -> ux_system_otg_device_type
```

Its values can be:

- UX_OTG_DEVICE_A
- UX_OTG_DEVICE_B
- UX_OTG_DEVICE_IDLE

A USB OTG host device can always ask for a role swap by issuing the command:

```
/* Ask the stack to perform a HNP swap with the device.  We relinquish the
   host role to A device. */
ux_host_stack_role_swap(storage -> ux_host_class_storage_device);
```

For a slave device, there is no command to issue but the slave device can set a state to change the role which will be picked up by the host when it issues a GET_STATUS and the swap will then be initiated.

```
/* We are a B device, ask for role swap.  The next GET_STATUS from the host
   will get the status change and do the HNP. */
ux_system_otg -> ux_system_otg_slave_role_swap_flag =
                                            UX_OTG_HOST_REQUEST_FLAG;
```

Index

Asix class, 4
audio class, 4
bulk in, 47, 49
bulk out, 47, 49
callback, 42, 51, 52, 55, 58
CDC-ACM class, 4
class instance, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 25, 27, 28, 29, 30, 32, 34, 36, 37, 38, 39, 40, 42, 43, 53, 56
configuration descriptor, 49
device descriptor, 49
device side, 44, 45
DPUMP, 44, 45, 46, 47
endpoint descriptor, 49
generic serial class, 4
handle, 25, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 51
HID class, 4
host side, 45
interface descriptor, 49
memory insufficient, 8, 20, 21, 24, 26, 27, 28, 29, 30, 32, 34, 36, 37, 40
OTG, 57, 58, 59
Picture Transfer Protocol, 48
PIMA class, 48, 49, 54
power management, 58
printer class, 4
prolific class, 4
receive thread, 45
semaphore, 59
slave, 58, 59
storage class, 4
target, 53
transmit thread, 45
USB host stack, 54
USBX pictbridge, 48
VBUS, 57, 58