MCCI Corporation
3520 Krums Corners Road
Ithaca, New York 14850 USA
Phone +1-607-277-1029
Fax +1-607-277-6844
www.mcci.com

MCCI OTG User's Guide

Engineering Report 950000681
Rev. D
Date: 2011/09/30

PROPRIETARY NOTICE AND DISCLAIMER

Unless noted otherwise, this document and the information herein disclosed are proprietary to MCCI Corporation, 3520 Krums Corners Road, Ithaca, New York 14850 ("MCCI"). Any person or entity to whom this document is furnished or having possession thereof, by acceptance, assumes custody thereof and agrees that the document is given in confidence and will not be copied or reproduced in whole or in part, nor used or revealed to any person in any manner except to meet the purposes for which it was delivered. Additional rights and obligations regarding this document and its contents may be defined by a separate written agreement with MCCI, and if so, such separate written agreement shall be controlling.

The information in this document is subject to change without notice, and should not be construed as a commitment by MCCI. Although MCCI will make every effort to inform users of substantive errors, MCCI disclaims all liability for any loss or damage resulting from the use of this manual or any software described herein, including without limitation contingent, special, or incidental liability.

MCCI, TrueCard, TrueTask, MCCI Catena, and MCCI USB DataPump are registered trademarks of MCCI Corporation.

MCCI Instant RS-232, MCCI Wombat and InstallRight Pro are trademarks of MCCI Corporation.

All other trademarks and registered trademarks are owned by the respective holders of the trademarks or registered trademarks.

NOTE: The code sections presented in this document are intended to be a facilitator in understanding the technical details. They are for illustration purposes only, the actual source code may differ from the one presented in this document.

Document Release History

Rev. A	2008/01/29	Original release
Rev. B	2008/04/11	Added client notificaiton API
Rev. C	2011/04/04	Changed document numbers to nine digit versions. DataPump 3.0 Updates.
Rev. D	2011/09/30	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

1 Introduction

1.1 Purpose

This document describes the APIs and user guide to support the HNP and SRP in the MCCI USB Embedded Host / On-The-Go USB host stack.

1.2 Scope

This document describes the API as used by the client applications. This document assumes familiarity with the HNP/SRP protocols and MCCI USB Embedded Host/OTG product. For more information on our USB Embedded Host/ OTG product see [DPREF].

1.3 Glossary

A-device	A device with a Standard-A or Micro-A plug inserted into its receptacle. The A-device supplies power to VBUS and is host at the start of a session.
B-device	B device with a Standard-B, Micro-B, or Mini-B plug inserted into its receptacle, or a captive cable ending in a Standard-A plug. The B-device is a peripheral at the start of a session.
EH	Embedded Host
HCD	*See* Host Controller Driver
Host controller	The hardware module responsible for operating the USB bus as a host.
Host Controller Driver	The software component that provides low–level access to the specific Host Controller in use. This term may refer to a specific instance of the software that models the host controller to upper layers of software, or it may refer to the entire collection of code that implements the driver. Where necessary, we refer to the collection of code as the "HCD Class", and the specific data structures and methods that represent a given instance as an "HCD Instance".
HNP	Host Negotiation Protocol
OTG	Abbreviation for USB On-The-Go
OTGCD	*See* OTG Controller Driver
OTG Controller	The hardware module responsible for operating a dual-role OTG connection.
OTG Controller	The software component that provides low-level access to a USB bus via an

Driver	OTG Controller. Normally export three APIs, an HCD API, a DCD API, and a (shared) OTG
Phy	Short for "physical layer". Often used as short-hand for "transceiver". MCCI uses this in the abbreviations for the API operatons that are used for accessing the phy.
SRP	Session Request Protocol
Transceiver	The hardware module responsible for low-level signaling on the USB bus.
URC	MCCI USB Resource Compiler
USBD	USB Driver, the generic term for the USB Management module.
USBDI	USB Driver Interface, the generic term for the API between USB function drivers and USBD.

1.4 Referenced Documents

[USBDI] *MCCI DataPump USBDI*, MCCI Engineering report 950000325

[HCDAPI] *MCCI DataPump HCD API,* MCCI Engineering report 950000324

[USBCORE] *On-The-Go and Embedded Host Supplement to the USB 2.0 Specification*, version 2.0. This specification is available on the World Wide Web site http://www.usb.org/.

[USBRC] *USBRC User's Guide,* MCCI Engineering Report 950000061

[DPREF] *MCCI USB DataPump User's Guide*, MCCI Engineering Report 950000066

2 OTG Support

OTG products are required to support HNP as an A-device. OTG products must support HNP as a B-device if they can support any OTG product as a peripheral.

2.1 OTG Descriptor

The B-device that supports HNP or SRP must respond by providing this descriptor in response to a GetDescriptor(Configuration) command.

The application must add the OTG keywords in the URC file to support HNP and/or SRP as B-device.

- SRP support: This is not used by the A-device during normal operation. However, this is used during compliance testing to automatically detect the capabilities of the B-device

  on-the-go srp;

- SRP and HNP support: If B-device supports HNP it must support SRP too.

  on-the-go srp hnp;

```
USB-resource-file 1.00 =
{
Device  {
    USB-version 2.0
    Class       0x00
    SubClass0x00
    Protocol0x00#none specified.
    Control-Packet-Size 64
    Vendor      0x040E  %tag% S_VENDOR_ID    # S_MCCI
    Product-ID  0xF909  %tag% S_PRODUCT_ID
    Device-Version  1.0
    serial-number   S_SERIALNUMBER
    } %no external name% ;
on-the-go srp hnp;
#### here is the first (and only) configuration.
Configuration 1
```

2.2 Request HNP

HNP is applicable only if an HNP capable B-device is attached to the OTG port. The notification would be sent to the client to indicate if HNP is capable or not when B-device is attached to the OTG port. The client application should send the following IOCTL to initiate the HNP.

Table 1 USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION

IOCTL	**USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION**
Function	Notify USBDI (via a port) that the function driver is through working with the port.
Description	Sent to USBDI via a USBPUMP_USBDI_PORT to notify USBDI that the function driver instance is through working with the device. This causes the driver to be detached from the device. After this IOCTL completes, USBDI will park the instance and will cease using the device. If the device is on a root OTG hub, and HNP is enabled and sensible, then USBDI will attempt to hand host control over to the other device
Input	USBPUMP_USBDI_FUNCTION *pFunction; The issuing function. USBPUMP_USBDI_PORT *pPort; The governing port.

IOCTL	USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION
Output	None
Return	USBPUMP_IOCTL_RESULT_SUCCESS for success
Notes	This is a syncronous IOCTL; but note that the actual OTG functionality might be remotely located, so it may take a while for this actually to have effect.

```
/* Setup Macro */
USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION_ARG Arg;
USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION_ARG_SETUP_V1(
          &Arg,
          (USBPUMP_USBDI_PORT *)pPortObject,
          NULL
          );
/* Send IOCTL */
(VOID) UsbPumpObject_Ioctl(
          pPortObject,
          USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION,
          &Arg,
          NULL
          );
```

The USBD will attempt HNP after suspending the port only if the port is HNP capable. The USBD will send the notification when HNP failed in the following cases.

- B-device is not disconnected within 200 ms after suspending port.
- B-device requests RESUME while suspending port.

2.3 Request SRP

The client application should send the following IOCTL to initiate SRP when VBUS is not present on the bus.

Table 2 USBPUMP_IOCTL_DCD_SESSION_REQUEST

IOCTL	USBPUMP_IOCTL_DCD_SESSION_REQUEST
Function	Ask the transceiver state machine to do an SRP if appropriate.
Description	This IOCTL tells the lower layers that the system software wants to run a B-bus session, but doesn't want to be host. This will cause SRP, etc., as needed. This is a one-shot; the FSM will automatically reset this request after initiating SRP. Although this is defined by the OTG specification, it can be used on non "OTG" systems (i.e., things that are only peripherals). So it's part of the core definition set; but it's usually only

IOCTL	**USBPUMP_IOCTL_DCD_SESSION_REQUEST**
	implemented in OTG systems.
Input	pInParam always NULL, pOutParam always NULL
Output	None
Return	USBPUMP_IOCTL_RESULT_SUCCESS for success -- which will always be the result unless there's some kind of system plumbing problem. The SRP (if any) proceeds asynchronously.
Notes	This IOCTL has no effect if the device is not a B-device. SRP should not be attempted unless the device has an OTG descriptor that says the device actually does SRP. This IOCTL should only be issued by the DataPump core (which is supposed to know whether the device is allowed to do SRP.

The client application needs to send down this IOCTL to the USBPUMP_USBPHY object. The USBD will notify the SRP_FAILED event when A-device is not supplying VBUS within 6 seconds. The client application should display a device not connected or not responding type error message when SRP failed in order to pass the USB-IF OPT FS B-device testing.

```
MY_PLATFORM_CONTEXT * CONST pMyPlatform = pPlatform->upf_pContext;
USBPUMP_USBPHY * CONST pUsbPhy
              =(USBPUMP_USBPHY *)pMyPlatform->upf_pUsbPhy;
(VOID) UsbPumpObject_Ioctl(
              &pUsbPhy->ObjectHeader,
              USBPUMP_IOCTL_DCD_SESSION_REQUEST,
              NULL,
              NULL
              );
```

2.4 SRP-based A-device

The MCCI host stack is the insertion-based by default. The host provides VBUS when the Transceiver is running in host mode and waits for a device to connect. In the SRP-based mode the A-device does not provide VBUS until B-device requests a session through SRP. So the A-device does not enumerate B-device if it is not SRP-capable device. The client application can change the host mode by sending the following IOCTL at any time. Changing mode from the insertion-based to the SRP-based would take effect when B-device is attached to the OTG port.

Table 3 USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED

IOCTL	**USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED**
Function	Set SRP based or Insertion based mode of OTG device when in an A-Role.

IOCTL	**USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED**
Description	This IOCTL directly controls the state transition between a_idle and a_wait_vrise. On the SRP based mode, FSM will transit from a_idle state to a_wait_vrise state only if a_srp_det is TRUE.
Input	This IOCTL takes a pointer to a paramter of type USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED_ARG, containing the following fields. BOOL fEnable; If logically TRUE, the FSM will act as SRP based OTG device. If it is FALSE, the FSM will act as insertion based OTG device.
Output	None
Return	USBPUMP_IOCTL_RESULT_SUCCESS for success -- which will always be the result unless there's some kind of system plumbing problem.
Notes	This IOCTL will not be immediate effect if the device is already in an a_host state.

The client application needs to send down this IOCTL to the USBPUMP_USBPHY object.

```
MY_PLATFORM_CONTEXT * CONST pMyPlatform = pPlatform->upf_pContext;
USBPUMP_USBPHY * CONST pUsbPhy
              =(USBPUMP_USBPHY *)pMyPlatform->upf_pUsbPhy;
USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED_ARG   IoctlArg;

USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED_ARG_SETUP_V1(&IoctlArg, TRUE);
(VOID) UsbPumpObject_Ioctl(
              &pUsbPhy->ObjectHeader,
               USBPUMP_IOCTL_OTGCD_ENABLE_SRP_BASED_ARG,
              &IoctlArg,
              NULL
              );
```

3 Client Notification

Annunciator objects provide a way to send notifications to the registered clients in the same task, or same processes. They broadcast notifications to any clients that have registered to receive notifications in a specific group of notifications; e.g., OTG notifications. The detail description of the Annunciator objects are illustrated in 950000686-(MCCI-Notification-Annunciator-Specification).doc

3.1 OTG Annunciator

The client application should obtain a session from the OTGFSM Annunciator object and register the session handler to receive the following OTGFSM event notifications.

Table 4 OTGFSM Event Notification

Notification ID	Notification Information
USBPUMP_OTGFSM_EVENT_START_SRP	B-device starts SRP USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_SRP_FAIL	A-device is not responding on SRP USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_SRP_SUCCESS	B-device is connected to A-device USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_OVER_POWER	A-device is in “a_vbus_err” state USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_NOT_SUPPORT	Unsupported Device USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_TRYING_HNP	A-device starts HNP USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_HNP_SUCCESS	A-device becomes a device and B-device becomes a host. USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_HNP_FAIL	A-device failed to change to a device role, or B-device failed to change to a host role. USBPUMP_OTGFSM * pOtgFsm
USBPUMP_OTGFSM_EVENT_DETECT_SRP	A-device detects SRP from B-device USBPUMP_OTGFSM * pOtgFsm

The client should enumerate the OTGFSM Annunciator object and call UsbPumpObject_OpenSession to obtain a session handler. If the client obtains a valid session handler in the callback function the client should register it in the Annunciator object.

```
#include "usbpump_annunciator.h"
#include "otgfsm.h"
struct __TMS_STRUCTNAME   (USBPUMP_NOTIFICATION_CLIENT_DATA)   {
          __TMS_USBPUMP_OBJECT_HEADER          ObjectHeader;
          __TMS_UPLATFORM *                            pPlatform;

          /* Annunciator session handler */
          __TMS_USBPUMP_SESSION_HANDLE         hOtgAnnunciatorSession;
```

```
          /* Annunciator object methods to be called by Observer to register
          || or unregister session handler to the Annunciator */
          __TMS_USBPUMP_ANNUNCIATOR_OBSERVER_INCALL OtgAnnunciatorInCall;

          /* client funtion to receive the notification */
          __TMS_USBPUMP_ANNUNCIATOR_OBSERVER_OUTCALL   OtgAnnunciatorOutCall;
          };

/* callback function for UsbPump_OpenSession */
static USBPUMP_API_OPEN_CB_FN
          OtgAnunciator_OpenSession_Callback;
static USBPUMP_ANNUNCIATOR_NOTIFICATION_FN
          Otg_ReceiveNotification;
```

The following code enumerates the Annunciator object and attempt to open a session.

```
USBPUMP_NOTIFICATION_CLIENT_DATA * pClient;
USBPUMP_OBJECT_ROOT *              pRootObject;
USBPUMP_OBJECT_HEADER *            pAnnunciatorObjectHeader;

pClient->pPlatform = pPlatform;

/*
|| set up USBPUMP_ANNUNCIATOR_OBSERVER_OUTCALL with
|| USBPUMP_ANNUNCIATOR_NOTIFICATION_FN to process the OTGFSM event
*/
USBPUMP_ANNUNCIATOR_OBSERVER_OUTCALL_SETUP_V1(
          &pClient->OtgAnnunciatorOutCall,
          Otg_ReceiveNotification
          );

pAnnunciatorObjectHeader = NULL;
while ((pAnnunciatorObjectHeader
                  = UsbPumpObject_EnumerateMatchingNames(
                        &pPumpRoot->Header,
                        pAnnunciatorObjectHeader,
                         USBPUMP_OTGFSM_ANNUNCIATOR_OBJECT_NAME
                        )) != NULL)
              {
              /*
              || Attempt to open a session with the Observer GUID. The client should
              || provide callback function to receive the sesseion handler
              */
              UsbPumpObject_OpenSession(
                  pAnnunciatorObjectHeader,
                  UsbPumpPlatform_Malloc(
                      pPlatform,
                      sizeOpenSessionRequestMemory
```

```
                    ),
                sizeOpenSessionRequestMemory,
                OtgAnunciator_OpenSession_Callback,
                pClient,
                &gk_UsbPumpOtgFsmAnnunciator_ObserverGuid,
                NULL,       /* pClientObject -- OPTIONAL */
                &pClient->OtgAnnunciatorInCall.GenericCast,
                sizeof(pClient->OtgAnnunciatorInCall),
                pClient,/* pClientHandle */
                &pClient->OtgAnnunciatorOutCall.GenericCast,
                sizeof(pClient->OtgAnnunciatorOutCall)
                );
            }
```

The following callback function checks if session is successfully opened and register pSessionHandle to the Annunciator object.

```
static VOID
OtgAnunciator_OpenSession_Callback(
          VOID *  pClientContext,
          USBPUMP_SESSION_HANDLE   SessionHandle,
           USBPUMP_API_STATUS  Status,
          VOID *  pOpenRequestMemory,
          RECSIZE sizeOpenRequestMemory
          )
          {
          USBPUMP_NOTIFICATION_CLIENT_DATA * CONST pClient = pClientContext;
          UINT32 retVal = 0;

          if (Status == USBPUMP_ANNUNCIATOR_STATUS_OK)
              {
              pClient->hOtgAnnunciatorSession = SessionHandle;
              }
          else
              {
              TTUSB_OBJPRINTF((
                  &pClient->ObjectHeader,
                  UDMASK_ERRORS,
                  " OtgAnnunciator_OpenSession_Callback:"
                  " OpenSession failed %x\n",
                  Status
                  ));
              }

          if (pOpenRequestMemory)
              {
              UsbPumpPlatform_Free(
                  pClient->pPlatform,
                  pOpenRequestMemory,
```

```
                sizeOpenRequestMemory
                );
            }

        /*
        || Client register the session handler using
        || USBPUMP_ANNUNCIATOR_REGISTER_NOTIFICATION_FN provided by the Annuciator
        */
        If (pClient->OtgAnnunciatorSession &&
            pClient->OtgAnnunciatorInCall.Observer.pRegisterFn)
            {
            retVal = (*pClient->OtgAnnunciatorInCall.Observer.pRegisterFn)(
                    pClient->hOtgAnnunciatorSession
                    );

            if (retVal != USBPUMP_ANNUNCIATOR_STATUS_OK)
                {
                TTUSB_OBJPRINTF((
                    &pClient->ObjectHeader,
                    UDMASK_ERRORS,
                    " OtgAnnunciator_OpenSession_Callback:"
                    " Registration failed\n"
                    ));
                }
            }
        }
```

The Annunciator object calls the following function to send the event notification with the additional notification information.

```
static VOID
Otg_ReceiveNotification(
        VOID *                  pClientHandle,
        USBPUMP_ANNUNCIATOR_NOTIFICATION NotificationId,
        CONST VOID *            pNotificationInfo,
        BYTES                   sizeNotificationInfo
        )
        {
        TTUSB_DEBUG(
            USBPUMP_NOTIFICATION_CLIENT_DATA * pClient = pClientHandle;
            )
        USBPUMP_OTGFSM * pOtgFsm = (USBPUMP_OTGFSM *)pNotificationInfo;
        USBPUMP_DEBUG_PARAMETER(pClientHandle);
        USBPUMP_UNREFERENCED_PARAMETER(sizeNotificationInfo);

        switch(NotificationId)
            {
        case USBPUMP_OTGFSM_EVENT_HNP_FAIL:
            if (pOtgFsm->Vars.otg_state == OTGST_b_wait_acon)
```

```
            {
            TTUSB_OBJPRINTF((
                &pClient->ObjectHeader,
                UDMASK_ANY | UDMASK_ERRORS,
                " Otg_ReceiveNotification: "
                "HNP Failed: Device Not Responding\n"
                ));
            }
        else
            {
            TTUSB_OBJPRINTF((
                &pClient->ObjectHeader,
                UDMASK_ANY | UDMASK_ERRORS,
                " Otg_ReceiveNotification: "
                "HNP Failed\n"
                ));
            }
        break;
    default:
        TTUSB_OBJPRINTF((
            &pClient->ObjectHeader,
            UDMASK_ANY,
            " \nOtg_ReceiveNotification: %s\n",
            UsbPumpOtgFsm_EventName(NotificationId)
            ));
        break;
        }
    }
```

3.2 USBD Annunciator

The client application should obtain a session from the USBD Annunciator object and register the session handler to receive the following USBD event notifications.

Table 5 USBD Event Notification

Notification ID	Notification Information
USBPUMP_USBDI_EVENT_DEVICE_MATCHED	Attached device is enumerated USBPUMP_USBDI_EVENT_DEVICE_MATCHED_INFO
USBPUMP_USBDI_EVENT_DEVICE_NOT_MATCHED	Attached device is not in target list USBPUMP_USBDI_EVENT_DEVICE_NOT_MATCHED_INFO
USBPUMP_USBDI_EVENT_DEVICE_DISABLED	USBPUMP_IOCTL_USBDI_PORT_IDLE_FUNCTION is called.

Notification ID	Notification Information
	USBPUMP_USBDI_EVENT_DEVICE_DISABLED_INFO
USBPUMP_USBDI_EVENT_PORT_OVER_CURRENT	GetPortStatus() returns PORT_OVER_CURRENT error USBPUMP_USBDI_EVENT_PORT_OVER_CURRENT_INFO
USBPUMP_USBDI_EVENT_HUB_DOUBLE_BUS_POWER	New hub instance is the second hub in a chain of bus-powered hubs. USBPUMP_USBDI_EVENT_HUB_DOUBLE_BUS_POWER_INFO
USBPUMP_USBDI_EVENT_HUB_NO_POWER	Hub has not enough power to operate. USBPUMP_USBDI_EVENT_HUB_NO_POWER_INFO
USBPUMP_USBDI_EVENT_HUB_OVER_CURRENT	GetHubStatus() returns HUB_OVER_CURRENT error USBPUMP_USBDI_EVENT_HUB_OVER_CURRENT_INFO
USBPUMP_USBDI_EVENT_HUB_LOCAL_POWER	Report a HubLocalPower state change USBPUMP_USBDI_EVENT_HUB_LOCAL_POWER_INFO
USBPUMP_USBDI_EVENT_TREE_TOO_DEEP	The device tree is too deep (bTier > 5) USBPUMP_USBDI_EVENT_TREE_TOO_DEEP_INFO
USBPUMP_USBDI_EVENT_DRIVER_LAUNCHED	The port is successfully bound to a driver USBPUMP_USBDI_EVENT_DRIVER_LAUNCHED_INFO
USBPUMP_USBDI_EVENT_DRIVER_LAUNCHED_FAILED	Failed to launch the driver on the port. USBPUMP_USBDI_EVENT_DRIVER_LAUNCHED_FAILED_INFO
USBPUMP_USBDI_EVENT_DRIVER_COMPLETE	The port is unbound from a driver USBPUMP_USBDI_EVENT_DRIVER_COMPLETE
USBPUMP_USBDI_EVENT_DEVICE_VANISHED	The device is detached from a port USBPUMP_USBDI_EVENT_DEVICE_VANISHED
USBPUMP_USBDI_EVENT_ENUMERATION_FAILURE	enumeration failed USBPUMP_USBDI_EVENT_ENUMERATION_FAILURE_INFO
USBPUMP_USBDI_EVENT_TIMEOUT	enumeration failed after attempting three times USBPUMP_USBDI_EVENT_ENUMERATION_FAILURE_IN

Notification ID	Notification Information
	FO
USBPUMP_USBDI_EVENT_INTERNAL_FAILURE	An internal failure occurrs during enumeration USBPUMP_USBDI_EVENT_INTERNAL_FAILURE_INFO
USBPUMP_USBDI_EVENT_CHANGE_POWER	Send any change in wMaxPower USBPUMP_USBDI_EVENT_CHANGE_POWER_INFO

The client should enumerate the USBD Annunciator object and call UsbPumpObject_OpenSession to obtain a session handler. If the client obtains a valid session handler in the callback function the client should register it in the Annunciator object.

```
#include "usbpump_annunciator.h"
#include "usbpump_usbdi_event.h"

struct __TMS_STRUCTNAME   (USBPUMP_NOTIFICATION_CLIENT_DATA)
          {
          __TMS_USBPUMP_OBJECT_HEADER          ObjectHeader;
          __TMS_UPLATFORM *                    pPlatform;

          /* Annunciator session handler */
          __TMS_USBPUMP_SESSION_HANDLE         hUsbdAnnunciatorSession;

          /* HNP capable port object for testing HNP */
          __TMS_USBPUMP_USBDI_PORT *           pHnpPort;

          /*
          || Annunciator object methods to be called by Observer to register
          || or unregister session handler to the Annunciator
          */
          __TMS_USBPUMP_ANNUNCIATOR_OBSERVER_INCALL UsbdAnnunciatorInCall;

          /* client funtion to receive the notification */
          __TMS_USBPUMP_ANNUNCIATOR_OBSERVER_OUTCALL   UsbdAnnunciatorOutCall;
          };

/* callback function for UsbPump_OpenSession */
static USBPUMP_API_OPEN_CB_FN
          UsbdAnunciator_OpenSession_Callback;
static USBPUMP_ANNUNCIATOR_NOTIFICATION_FN
          Usbd_ReceiveNotification;
```

The client should use USBPUMP_USBDI_USBD_CONFIG_INIT_V4 to specifiy the maximum number of the USBD Annunciator sessions.

```
static CONST USBPUMP_USBDI_USBD_CONFIG sk_UsbPumpUsbd_Config =
```

```
          USBPUMP_USBDI_USBD_CONFIG_INIT_V4(
              NULL, 0, 0, 0, 0,
              USBPUMP_USB20_TATTDB_DEFAULT, USBPUMP_USB20_TRSTRCY_DEFAULT,
              USBPUMP_USB20_TDSETADDR_DEFAULT, USBPUMP_USB20_TSETADDRRCY_DEFAULT,
              USBPUMP_USB20_TDRQCMPLTND_DEFAULT, USBPUMP_USB20_TDRETDATA1_DEFAULT,
              USBPUMP_USB20_TDRETDATAN_DEFAULT, USBPUMP_USB20_TRSMRCY_DEFAULT,
              21, 7, NULL,
              4 /* Maximum number of the USBD Annunciator session */,
              UDMASK_CHAP9 | UDMASK_USBDI | UDMASK_HUB);
```

The following code enumerates the Annunciator object and attempt to open a session.

```
USBPUMP_NOTIFICATION_CLIENT_DATA * pClient;
USBPUMP_OBJECT_ROOT *              pRootObject;
USBPUMP_OBJECT_HEADER *            pAnnunciatorObjectHeader;

pClient->pPlatform = pPlatform;

/*
|| set up USBPUMP_ANNUNCIATOR_OBSERVER_OUTCALL with
|| USBPUMP_ANNUNCIATOR_NOTIFICATION_FN to process the USBD event
*/
USBPUMP_ANNUNCIATOR_OBSERVER_OUTCALL_SETUP_V1(
          &pClient->UsbdAnnunciatorOutCall,
          Usbd_ReceiveNotification
          );

pAnnunciatorObjectHeader = NULL;
while ((pAnnunciatorObjectHeader
                  = UsbPumpObject_EnumerateMatchingNames(
                        &pPumpRoot->Header,
                        pAnnunciatorObjectHeader,
                        USBPUMP_USBDI_ANNUNCIATOR_OBJECT_NAME
                        )) != NULL)
              {
              || Attempt to open a session with the Observer GUID. The client should
              || provide callback function to receive the sesseion handler
              */
              UsbPumpObject_OpenSession(
                  pAnnunciatorObjectHeader,
                  UsbPumpPlatform_Malloc(
                      pPlatform,
                      sizeOpenSessionRequestMemory
                      ),
                  sizeOpenSessionRequestMemory,
                  UsbdAnunciator_OpenSession_Callback,
                  pClient,
                  &gk_UsbPumpUsbdAnnunciator_ObserverGuid,
                  NULL,       /* pClientObject -- OPTIONAL */
```

```
                &pClient->UsbdAnnunciatorInCall.GenericCast,
                sizeof(pClient->UsbdAnnunciatorInCall),
                pClient,/* pClientHandle */
                &pClient->UsbdAnnunciatorOutCall.GenericCast,
                sizeof(pClient->UsbdAnnunciatorOutCall)
                );
            }

static VOID
UsbdAnunciator_OpenSession_Callback(
          VOID *  pClientContext,
          USBPUMP_SESSION_HANDLE   pSessionHandle,
          USBPUMP_API_STATUS   Status,
          VOID *  pOpenRequestMemory,
          RECSIZE sizeOpenRequestMemory
          )
          {
          * CONST pClient = pClientContext;
          UINT32 retVal = 0;

          if (Status == USBPUMP_ANNUNCIATOR_STATUS_OK)
              {
              pClient->hUsbdAnnunciatorSession = SessionHandle;
              }
          else
              {
              TTUSB_OBJPRINTF((
                  &pClient->ObjectHeader,
                  UDMASK_ERRORS,
                  " UsbdAnnunciator_OpenSession_Callback:"
                  " OpenSession failed %x\n",
                  Status
                  ));
              }

          if (pOpenRequestMemory)
              {
              UsbPumpPlatform_Free(
                  pClient->pPlatform,
                  pOpenRequestMemory,
                  sizeOpenRequestMemory
                  );
              }

          /*
          || Client register the session handler using
          || USBPUMP_ANNUNCIATOR_REGISTER_NOTIFICATION_FN provided by the Annuciator
          */
          If (pClient->hUsbdAnnunciatorSession &&
```

```
            pClient->UsbdAnnunciatorInCall.Observer.pRegisterFn)
            {
            retVal = (*pClient->UsbdAnnunciatorInCall.Observer.pRegisterFn)(
                    pClient->hUsbdAnnunciatorSession
                    );

            if (retVal != USBPUMP_ANNUNCIATOR_STATUS_OK)
                {
                TTUSB_OBJPRINTF((
                    &pClient->ObjectHeader,
                    UDMASK_ERRORS,
                    " UsbdAnunciator_OpenSession_Callback:"
                    " Registration failed\n"
                    ));
                }
            }
        }

static VOID
Usbd_ReceiveNotification(
        VOID *                  pClientHandle,
        USBPUMP_ANNUNCIATOR_NOTIFICATION NotificationId,
        CONST VOID *            pNotificationInfo,
        BYTES                   sizeNotificationInfo
        )
        {
        USBPUMP_NOTIFICATION_CLIENT_DATA * CONST pClient = pClientHandle;

        USBPUMP_UNREFERENCED_PARAMETER(sizeNotificationInfo);

        switch(NotificationId)
            {
        case USBPUMP_USBDI_EVENT_DEVICE_MATCHED:
            {
            USBPUMP_USBDI_EVENT_DEVICE_MATCHED_INFO * CONST pInfo
                = pNotificationInfo;
            if (pInfo->fHnpCapable)
                {
                pClient->pHnpPort = pInfo->pPort;
                TTUSB_OBJPRINTF((
                    &pClient->ObjectHeader,
                    UDMASK_ANY | UDMASK_FLOW,
                    " Usbd_ReceiveNotification:"
                    " HNP device is attached: %p\n", pInfo->pPort
                    ));
                }
            }
            break;
        case USBPUMP_USBDI_EVENT_DEVICE_VANISHED:
```

```
    {
    USBPUMP_USBDI_EVENT_DEVICE_VANISHED_INFO * CONST pInfo
        = pNotificationInfo;
    if (pInfo->pPort == pClient->pHnpPort)
        {
        pClient->pHnpPort = NULL;
        TTUSB_OBJPRINTF((
            &pClient->ObjectHeader,
            UDMASK_ANY | UDMASK_FLOW,
            " Usbd_ReceiveNotification:"
            " HNP device has vanished: %p\n", pInfo->pPort
            ));
        }
    }
    break;

default:
    TTUSB_OBJPRINTF((
        &pClient->ObjectHeader,
        UDMASK_ANY,
        " Usbd_ReceiveNotification: %s\n",
        UsbPumpUsbd_EventName(NotificationId)
        ));
    break;
    }
}
```