ZIGBEE/USB ADAPTER U1(-Q)

REFERENCE MANUAL

ubisys

1. OVERVIEW

Thank you for purchasing this ubisys ZigBee USB/Adapter.
You have decided for a high-quality product with first-rate support!

This reference manual provides operating and maintenance instructions, interface specifications, command references and more. It is primarily intended for system integrators, not end-users.

If you have any questions or need additional support, please visit the support pages that best fit your background:

If you are a consumer (private household) or installer, please visit the Smart Home support pages at http://www.ubisys.de/en/smarthome/support.html for contact details.

As a commercial customer, please visit the Engineering support pages at http://www.ubisys.de/en/engineering/support.html for contact details.

2. CONTENTS

3. FEATURES

- ZigBee 3.0 Certified Product
- Supports all ZigBee device roles: Coordinator & Trust Center, Router, Non-Sleeping End-Device, Sleeping End-Device
- ZigBee Green Power Proxy with the ability to operate as ZigBee Green Power Combined Device, when the controlling application provides ZigBee Green Power Sink functionality
- Uniform API across Windows, Linux and macOS based on either WinUSB or libusb
- Provides protocol independent raw capture capabilities (MAC promiscuous mode), facilitating the design of sniffers, automated test equipment and similar solutions; can operate as a node on the ZigBee network and change back and forth between ZigBee and sniffer mode[1]
- Reliable, unattended, maintenance-free 24/7 operation (same solution used on ubisys Gateway G1)
- Made in Germany using high-quality, enduring parts for many years of life expectancy
- USB 2.0 full-speed device
- On-board inverted-F antenna
- Power supply: 5V=, 50mA (bus-powered), low power dissipation: 0.3W
- ZigBee USB/Adapter U1: ARM7TDMI at 48MHz, 512KB Flash, 64KB RAM, Texas Instruments CC2520, 5dBm transmit power, -98dBm receiver sensitivity, ubisys IEEE 802.15.4 MAC
- ZigBee USB/Adapter U1-Q: Cortex-M4 at 48MHz, 512KB Flash, 128KB SRAM, Qorvo GP712, RFX2411, ?dBm transmit power, -?dBm receiver sensitivity, Qorvo IEEE 802.15.4 MAC
- ubisys ZigBee stack for best-in-class reliability and performance
- Supports all channels in the 2.4 GHz band, i.e. channels 11-26 as per IEEE 802.15.4:
  Primary = { 11, 15, 20, 25 }; Secondary = { 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24, 26 }
- Supports joining centralized and distributed security networks as router
- Supports forming simple centralized security networks as Coordinator and Trust Center
- Supports forming distributed security networks as router
- Three pre-configured Trust Center Link-Keys for joining:
  - Global Default Trust Center Link-Key (“ZigBeeAlliance09”)
  - ZigBee 3.0 Global Distributed Security Link-Key[2]
  - Device-individual link-key derived from installation code – also printed as text and QR barcode
- Extended neighbour table with up to 78 entries for routers and end-devices -
  more than three times the capacity required by the standard (25)
- Extended routing table with up to 96 entries for ad hoc and many-to-one routes -
  nearly ten times the capacity required by the standard (10)
- Extended buffering for sleeping end-devices with up to 24 buffers -
  24 times the capacity required by the standard (1)
- Extended APS duplicate rejection table with up to 64 slots -
  64 times the capacity required by the standard (1)
- Extensive transmit and receive queues for optimum through-put and minimum packet drop rate
- Reliable and scalable network-wide broadcasts featuring passive acknowledgments
- Reliable packet forwarding with automatic network-level retries
- Very sophisticated routing algorithm for reliable ad hoc routing – avoids routing loops even in case of concurrent route requests with overlapping source/destination
- Firmware upgradable via USB in the field
- Flame retardant housing (V-0); black, RAL 9005
- OEM and design-in version available upon request, e.g. for professional gateways (c.f. ubisys G1)

[1] Since application firmware 1.70
[2] Since application firmware 1.68. Prior, pre-certification key (D0:..:DF)

4.1. Anatomy of the Device

The U1 runs a specifically designed, proprietary, vendor-specific USB protocol. In addition to the control pipe (endpoint #0), which executes synchronous (blocking) request/response exchanges, there are also two asynchronous (non-blocking) data pipes, one for outbound transfers (endpoint #1, from host computer to U1), and another one for inbound transfers (endpoint #2, from U1 to host computer). Additionally, endpoint #3 provides the ability to raise interrupt requests to the host[3].

The protocol primitives facilitate implementation of a ZigBee Gateway Device (ZGD) according to the ZigBee Network Gateway Device Specification [R0], on top of the USB protocol.

[R0] ZigBee Network Device Gateway Specification, Document No. 07-5468-35.

4.2. Vendor and Product Identifiers

The Vendor ID (VID) for genuine ubisys products is 0x19A6.
The Product ID (PID) for U1(-Q) is 0x0004.

4.3. Control Transfers

TBD

4.4. Data Transfers

TBD

[3] This feature is currently not used.

5. 7BFX™ APPLICATION PROGRAMMING INTERFACE

Any USB host application framework, which supports custom USB devices, can interface to the U1. For Windows, Linux and macOS operating systems, ubisys provides the 7Bfx™ API, which greatly simplifies integration of the U1 into applications and solutions. While 7Bfx™ is a native C++ framework primarily designed for use within other C++ code, it can also be used from other programming languages like C# or plain C with a thin interworking layer.

Developers need to be familiar with IEEE 802.15.4 and ZigBee documents, in particular the ZigBee Gateway Device Specification, ZigBee Core Stack Specification, ZigBee Base Device Behavior Specification, ZigBee Cluster Library Specification, ZigBee Green Power Specification and further applicable device specifications, for example the ZigBee Home Automation Profile or ZigBee Lighting & Occupancy Specification. Many of the status codes mentioned in this document are status codes belonging to the IEEE 802.15.4 PHY, MAC, ZigBee NWK, APS, ZDO, ZCL or ZGD.

Important Notice:

Instead of interfacing directly with U1 using the 7Bfx™ API, which is presented in this document and provides a relatively low-level marshalled interface to ZigBee APS and ZDO, you might prefer higher-level interfaces to U1, which ubisys also provides.

Available options include the ubisys ZigBee Gateway Device Service (zgdd), which provides GRIP, a standards-compliant binary TCP/IP protocol based on ASN.1 and allows sharing a single U1 amongst different applications.

On top of zgdd, ubisys also offers the ubisys Smart Facility Service (facilityd), which completely relieves developers from commissioning and configuring ZigBee devices and comes with C/C++ and Java client SDKs. Adapters to IoTivity (www.github.com/ubisys/iotivity), Apple HomeKit, Amazon Alexa and other high-level smart home and IoT frameworks are also available.

Additional companion services complete the offering. For example, the ubisys ZigBee Over-the-Air Upgrade Service (otad) is a ready-to-use, fully-fledged firmware upgrade server. The ubisys ZigBee Time Server (ztimed) disseminates time from an internal real-time clock or internet NTP server to devices in the ZigBee network.

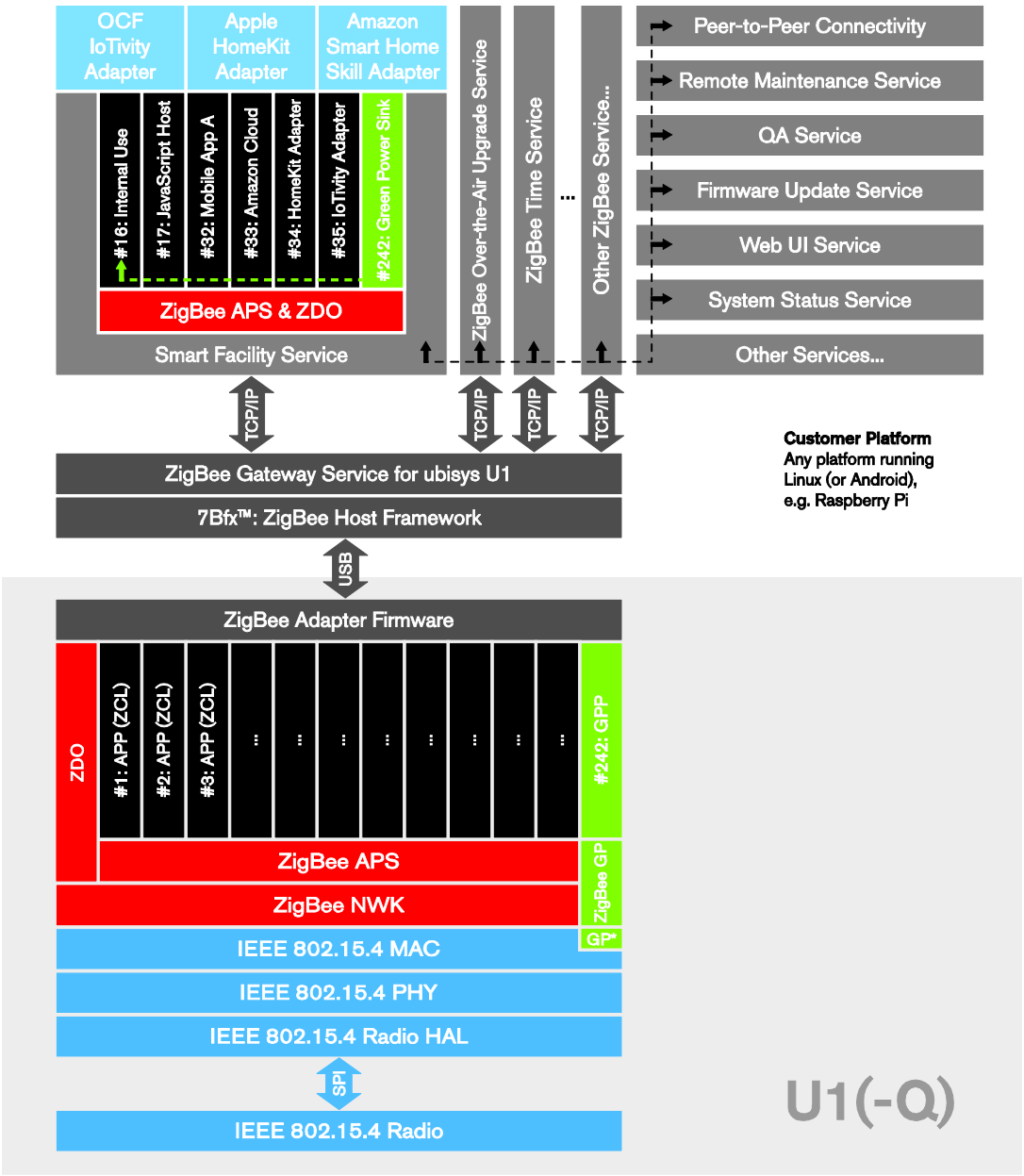

Figure 1: Example of an Overall System Architecture for a ZigBee Gateway based on U1(-Q)

5.1. API Reference

To use the 7Bfx API, `#include "u7bfx.h"` and link against the appropriate library for your target platform. On Windows, this would be the import library for either x86 or x64 processors, in the debug or release build. When using the Microsoft Visual C++ Compiler, there is no need to specifically link against one of these libraries, as the right one is automatically referenced by including u7bfx. Conversely, on Linux the 7Bfx API is provided in a static library for a range of processor architectures, including ARMv5 and ARMv7 and needs to be passed to the linker.

In C++, all symbols are defined in the u7bfx namespace and all API routines use the `__stdcall` calling convention and return a `HRESULT` value.

5.1.1. Genuine ZigBee Adapter Status Codes

Some of the APIs return genuine ZigBee Adapter status codes as listed below:

Name (Value)	Description
SUCCESS (0)	The operation completed successfully.
TIMEOUT (1)	The operation failed because a required action did not complete within the expected time frame
GENERAL_ERROR (2)	A general error occurred.
PARAMETER_MISSING (3)	The operation failed because a required parameter was missing.
PARAMETER_INVALID_VALUE (4)	The operation failed because a supplied parameter had an invalid value.
NETWORK_NOT_READY (5)	The operation failed because the network was not ready.
EMPTY (6)	
NOT_ALLOWED (7)	The adapter did not execute the operation because it was not allowed.
MEMORY_ERROR (8)	The operation failed because of insufficient memory.
APS_FAILURE (9)	The operation failed at the Application Support sublayer.
NETWORK_FAILURE (10)	The operation failed at the Network layer.

For convenience, u7bfx.h defines following enumeration values:

```
enum { statusSuccess, statusTimeout, statusGeneralError,
       statusParameterMissing, statusParameterInvalidValue,
       statusNetworkNotReady, statusEmpty, statusNotAllowed, statusMemoryError,
       statusAPSFailure, statusNetworkFailure };
```

5.1.2. Initialization and Shutdown

Before using any other function, you have to call `Initialize()`, which will return an instance handle upon success. When initialization succeeded and you terminate the application, you should always call `Shutdown()` and pass it the handle previously returned by `Initialize()`.

5.1.2.1. `Initialize()` – Initialize library

Declaration

```
HRESULT STDAPICALLTYPE Initialize(HANDLE &hInstance);
```

Synopsis

Initializes the 7Bfx library for use.

Arguments

Argument	Type (Direction)	Description
`hInstance`	`HANDLE` (out)	Reference to an instance handle variable, which receives the instance handle of the 7Bfx instance for this application, if the function succeeded.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

For example, this could be a typical application start-up sequence:

```
HANDLE hInstance, hDeviceList;

if (!AfxWinInit(GetModuleHandle(0), 0, GetCommandLine(), 0))
{
	std::cerr << "MFC initialization failed." << std::endl;
	return 1;
}

ENSURE(SUCCEEDED(u7bfx::Initialize(hInstance)));
```

5.1.2.2. `Shutdown()` – Shutdown library

Declaration

```
HRESULT STDAPICALLTYPE Shutdown(const HANDLE hInstance);
```

Synopsis

Performs a graceful shutdown of the 7Bfx library after use. Performs clean-up and returns resources to the operating system.

Arguments

Argument	Type (Direction)	Description
`hInstance`	`HANDLE` (in)	The instance handle as returned by a prior call to `Initialize()`.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Similar to the start-up sequence, the shutdown sequence should look like this:

```
VERIFY(SUCCEEDED(u7bfx::Shutdown(hInstance)));
```

5.1.3. Device Enumeration

5.1.3.1. `GetDeviceList()` – List all attached U1 adapters

Declaration

```
HRESULT STDAPICALLTYPE GetDeviceList(HANDLE &hDeviceList);
```

Synopsis

Creates a list of U1 adapters currently attached to the system. You can pass the returned handle to functions like `GetDeviceSerialNumber()` and `OpenDevice()`, for example.

Note: Always call `DestroyDeviceList()` for the returned handle after use.

Arguments

Argument	Type (Direction)	Description
`hDeviceList`	`HANDLE` (out)	Reference to a device list handle, which can be used to fetch information about devices, connect to devices, etc.

Return Value

A `HRESULT` conveying the success/failure of the operation and other information, like error codes in case of a failure and the number of devices in the list in case of success. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value and in case of success `HRESULT_CODE()` to determine the number of devices returned in the list.

Example

A typical enumeration sequence would look like this:

```
// Obtain a list of all ubisys ZigBee/USB adapters attached to the system
const HRESULT hResult = u7bfx::GetDeviceList(hDeviceList);

// If the call succeeded, the number of devices is available in the result
// code and we can iterate over individual devices in the list...
if (SUCCEEDED(hResult))
{
	std::cout << "Enumeration succeeded. " << HRESULT_CODE(hResult) <<
		" devices found:" << std::endl;

	for (int nDevice = 0; nDevice < HRESULT_CODE(hResult); nDevice++)
	{
		// Do something with the device
		...
	}
}

// Done with the list, release it
u7bfx::DestroyDeviceList(hDeviceList);
```

5.1.3.2. `DestroyDeviceList()` – Release device list

Declaration

```
HRESULT STDAPICALLTYPE DestroyDeviceList(const HANDLE hDeviceList);
```

Synopsis

Releases the resources required to keep the list of U1 adapters currently attached to the system.

Arguments

Argument	Type (Direction)	Description
`hDeviceList`	`HANDLE` (in)	A device list handle previously returned by a successful call to `GetDeviceList()`.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Refer to the example for `GetDeviceList()`, in section 5.1.3.1.

5.1.3.3. `GetDevicePath()` – Obtain device driver instance path

Declaration

```
HRESULT STDAPICALLTYPE GetDevicePath(const HANDLE hDeviceList,
      const unsigned int nDeviceIndex, const LPTSTR pszDeviceName,
      const unsigned int cbDeviceName);
```

Synopsis

Returns a device driver instance path on operating systems, which support and require such information to instantiate a device driver for a specific adapter connected to the system. The device need not be open to query its fully qualified path.

Arguments

Argument	Type (Direction)	Description
`hDeviceList`	`HANDLE` (in)	A device list handle previously returned by a successful call to `GetDeviceList()`.
`nDeviceIndex`	`unsigned int` (in)	Index to a specific device within the list of devices, starting at zero, for which the device path is requested
`pszDeviceName`	`LPTSTR` (in for location, out for data stored at this location)	Pointer to a string buffer, which receives the device path. The buffer must have at least `cbDeviceName` bytes of storage capacity.
`cbDeviceName`	`unsigned int` (in)	Size of the output string buffer, in bytes, for the device path.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`. This is an optional feature and platforms that do not support it might return `E_NOTIMPL`.

Example

Below example, when executed on a Windows Platform, would return a resulting fully qualified device path like `\\?\usb#vid_19a6&pid_0004#0000002568#{9651f68a-847b-4bb0-9c92-2d2bc71dc876}`. Notice you should not require this function to succeed on platforms that do not support it.

```
      TCHAR szPath[MAX_PATH + 1];

      // Determine the fully qualified device path, if available
      if (FAILED(GetDevicePath(hDeviceList, nIndex, szPath, sizeof(szPath))))
            return E_FAIL;
```

5.1.3.4. `GetDeviceSerialNumber()` – Obtain an adapter's USB serial number

Declaration

```
HRESULT STDAPICALLTYPE GetDeviceSerialNumber(const HANDLE hDeviceList,
      const unsigned int nDeviceIndex, const LPTSTR pszSerialNumber,
      const unsigned int cbSerialNumber);
```

Synopsis

Returns the serial number of an attached adapter, in the format `VVVV-PPPP-SSSSSSSSSS`, where `VVVV` is the four-digit ASCII hexadecimal representation of the USB Vendor ID (`19A6` = ubisys technologies GmbH), `PPPP` is the four-digit ASCII hexadecimal representation of the USB Product ID (`0004` = ubisys ZigBee/USB Adapter U1[4]). The device need not be open to query its serial number.

Arguments

Argument	Type (Direction)	Description
`hDeviceList`	`HANDLE` (in)	A device list handle previously returned by a successful call to `GetDeviceList()`.
`nDeviceIndex`	`unsigned int` (in)	Index to a specific device within the list of devices, starting at zero, for which the serial number is requested
`pszSerialNumber`	`LPTSTR` (in for location, out for data stored at this location)	Pointer to a string buffer, which receives the serial number. The buffer must have at least `cbDeviceName` bytes of storage capacity. A buffer of 21 bytes (including zero-termination character) is sufficient to store the resulting string.
`cbDeviceName`	`unsigned int` (in)	Size of the output string buffer, in bytes, for the serial number.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`. This is an optional feature and platforms that do not support it might return `E_NOTIMPL`.

Example

Below example would return a resulting string like `19a6-0004-0000002568`. If the application is showing serial numbers in a user interface, it should only display the trailing part. For consistency across different applications, the following convention should be used to format the serial number for display: `S/N: 0000002568`. This is how the device appears in the ubisys Network Manager software, for example.

```
      TCHAR szSerial[21];

      // Obtain the serial number string of the first attached adapter
      if (FAILED(GetDeviceSerialNumber(hDeviceList, 0, szSerial, sizeof(szSerial))))
            return E_FAIL;
```

[4] Notice the product ID would be the same for fully compatible devices implementing the same native USB protocol detailed in chapter 4, including the ubisys U1-Q, and the ubisys ZigBee Development Boards ZDB AT91SAM7S512+CC2520, ZDB ATSAM4S+CC2520, ZDB ATSAM4S+GP712, etc.

5.1.4. Adapter Device Management

5.1.4.1. `OpenDevice()` – Prepare an adapter for actual use

Declaration

```
HRESULT STDAPICALLTYPE OpenDevice(const HANDLE hDeviceList,
      const unsigned int nIndex, HANDLE &hDevice);
```

Synopsis

Prepares exclusive access to an adapter, such that the application can use it for subsequent data transmissions in the ZigBee network or for capturing raw MAC frames. The function fails if the device is already open, regardless whether another application or the same application opened the device before. Always call `CloseDevice()` for each handle returned by a successful call to `OpenDevice()`.

Arguments

Argument	Type (Direction)	Description
`hDeviceList`	`HANDLE` (in)	A device list handle previously returned by a successful call to `GetDeviceList()`.
`nDeviceIndex`	`unsigned int` (in)	Index to a specific device within the list of devices, starting at zero, for which the serial number is requested
`hDevice`	`HANDLE` (out)	Reference to a device list handle, which can be used to fetch information about devices, connect to devices, etc.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`. If the device was already open or the operation fails due to unavailable resources or the device having been removed from the system in the meantime, this function returns `E_FAIL`. Other error codes include `E_HANDLE` indicating the supplied device list was invalid, and `E_INVALIDARG` indicating the device index was out of bounds.

Example

Below example attempts to open the device with index `nDevice` in the device list.

```
      // Open the device
      if (FAILED(u7bfx::OpenDevice(hDeviceList, nDevice, hDevice)))
            throw std::runtime_error("failed to open adapter");

      // Use the device here
      ...

      // Close the device
      CloseDevice(hDevice);
```

5.1.4.2. `CloseDevice()` – Release device after use

Declaration

```
HRESULT STDAPICALLTYPE CloseDevice(const HANDLE hDevice);
```

Synopsis

Closes the connection to a specific U1 adapter when it is no longer required.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Refer to the example for `OpenDevice()`, in section 5.1.4.

5.1.4.3. `RevertToFactoryFreshSettings()` – Restore factory configuration

Declaration

```
HRESULT STDAPICALLTYPE RevertToFactoryFreshSettings(const HANDLE hDevice,
      const bool bFull = false);
```

Synopsis

Reverts the persistent storage in non-volatile memory to its factory-fresh state. A full factory reset results in exactly the same state as delivered from the factory, whereas a typical factory reset allows retaining some settings, in particular the ZigBee network layer outgoing security frame counter.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`bFull`	`bool` (in)	Set to true for a full factory reset, false to preserve certain recommended settings, in particular security frame counters. Default is false.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.4.4. `SetExtraInfo()` – Associate additional information with a device

Declaration

```
HRESULT STDAPICALLTYPE SetExtraInfo(const HANDLE hDevice, void *pExtraInfo);
```

Synopsis

Provides a way to associate a device instance handle with an application defined object, for example a device manager instance. This is a convenient way utility to facilitate instance pointer look-ups, for example when the application implements a C++ object for each managed ZigBee adapter device and the application needs to determine the object pointer given the handle. The stored pointer can be obtained via subsequent calls to `GetExtraInfo()`.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pExtraInfo`	`void *` (in)	An arbitrary, application-defined pointer or pointer-sized integer via cast to uint_ptr_t or int_ptr_t to be associated with a device instance handle.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Below example associates a C++ object using its implicit `this` pointer with a device handle.

```
    // Associate the device handle with a pointer to this manager object
    ENSURE(SUCCEEDED(u7bfx::SetExtraInfo(m_hDevice, this)));
```

5.1.4.5. `GetExtraInfo()` – Associate additional information with a device handle

Declaration

```
HRESULT STDAPICALLTYPE GetExtraInfo(const HANDLE hDevice, void *&pExtraInfo);
```

Synopsis

Obtains the application-defined information previously stored with `SetExtraInfo()`. When the request succeeds, the supplied storage will be set to the value previously provided by the application.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pExtraInfo`	`void *&` (in for location, out for data stored at this location)	Reference to a pointer or pointer-sized integer, which shall receive the extra information previously associated with this device handle using a call to `SetExtraInfo()`.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Below example obtains a C++ object given a device handle and uses this object to perform an action.

```
CMyDeviceManager *pDevice;

// Obtain the object associated with this device instance
ENSURE(SUCCEEDED(u7bfx::GetExtraInfo(hDevice,
       reinterpret_cast<void *&>(pDevice))));

// Sanity check: We expect this object to live on the heap
ASSERT(AfxIsMemoryBlock(pDevice, sizeof(CMyDeviceManager)));

// Sanity check: As we are casting from a void pointer,
// make sure the type matches
ASSERT(dynamic_cast< CMyDeviceManager *>(pDevice));

// Now do something with the manager object...
static_cast<CMyClass *>(pMyObject)->DoSomething();
```

5.1.4.6. `GetVersions()` – Obtain Application, Stack and Hardware Versions

Declaration

```
HRESULT STDAPICALLTYPE GetVersions(const HANDLE hDevice,
      unsigned int &dwFirmwareVersion, unsigned char &bStackVersion,
      unsigned char &bHardwareVersion);
```

Synopsis

Obtains the version numbers of ZigBee/USB adapter application firmware, Compact7B™ ZigBee Stack release and the underlying hardware platform.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwFirmwareVersion`	`unsigned int &` (out)	Reference to a 32-bit unsigned integer, which receives the firmware version. This version follows the OTA Upgrade cluster scheme, i.e. is organized as AAaaSSss, where: - AA (MSB) is the major application version - aa is the minor application version - SS is the major Compact7B™ ZigBee stack version - ss (LSB) is the minor Compact7B™ ZigBee stack version For example, firmware application version 1.70 embedding Compact7B™ ZigBee stack release version 1.86 is encoded as 0x01460156.
`bStackVersion`	`unsigned char &` (out)	Reference to an 8-bit unsigned integer, which receives the ZigBee specification version. This value is always 0 for existing devices.
`bHardwareVersion`	`unsigned char &` (out)	Reference to an 8-bit unsigned integer, which receives the hardware version, which is essentially the platform type and may include specific hardware revision information in the future: 0 – AT91SAM7S256 + CC2420 1 – ubisys U1, AT91SAM7S512 + CC2520 2 – ubisys U1-M, ATSAM4S8B + CC2520 3 – ZigBee Adapter provided by an embedded ZigBee Stack[5] 4 – ubisys U1-Q, ATSAM4S8B + GP712 + RFX2411

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

```
ENSURE  (SUCCEEDED(u7bfx::GetVersions(hDevice, dwFirmwareVersion,
      bStackVersion, bHardwareVersion)));

std::wcout << L"firmware version: " << (dwFirmwareVersion >> 24)
      << L'.' << std::setw(2) << std::setfill(L'0')
      << ((dwFirmwareVersion >> 16) & 0xff) << L", stack release: " << std::setw(0)
      << ((dwFirmwareVersion >> 8) & 0xff) << L'.' << std::setw(2) << std::setfill(L'0')
      << (dwFirmwareVersion & 0xff)
      << L", stack version: "
      << bStackVersion << L", hardware version: " << bHardwareVersion
      << std::endl;
```

[5] This is a virtual adapter, which exists in applications incorporating an embedded Compact7B™ ZigBee stack. There is no further information about the underlying hardware. Examples are the ubisys ZigBee Gateway Service (zgdd) on a Rapsberry Pi with Qorvo GP711 or GP712 radio, with or without additional radio front-end module.

5.1.4.7. `GetExtendedAddress()` – Obtain Adapter's IEEE EUI-64

Declaration

```
HRESULT STDAPICALLTYPE GetExtendedAddress(const HANDLE hDevice,
      unsigned long long &qwExtendedAddress);
```

Synopsis

Returns the adapter's 64-bit IEEE 802.15.4 MAC hardware address, also referred to as MACID or EUI-64. This address is a fixed, universally unique identifier permanently stored in the adapter and typically printed on the housing.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`qwExtendedAddress`	`unsigned long long &` (out)	Reference to a 64-bit unsigned integer, which receives the adapter's IEEE 802.15.4.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

```
ENSURE(SUCCEEDED(u7bfx::GetExtendedAddress(hDevice, qwExtendedAddress)));

std::wcout << L"IEEE extended address: " << std::hex << std::setw(16)
      << std::setfill(L'0') << qwExtendedAddress << std::endl;
```

5.1.4.8. `GetNetworkParameters()` – Obtain ZigBee Network Settings

Declaration

```
HRESULT STDAPICALLTYPE GetNetworkParameters(const HANDLE hDevice,
      unsigned long long &qwExtendedPANID, unsigned short &wPANID,
      unsigned short &wShortAddress, unsigned char &nChannel,
      bool &bAssociationPermit);
```

Synopsis

Returns the adapter's ZigBee network settings, for example the full 64-bit network identifier (extended PAN-ID), the 16-bit short network identifier used for addressing at the MAC level, the devices short address and operating channel. Some data is not valid before the device has started network operations. For example, a short address of 0xFFFF indicates that the device

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`qwExtendedPANID`	`unsigned long long &` (out)	Reference to a 64-bit unsigned integer, which receives the extended PAN-ID of the ZigBee network as advertised in the beacon payload, for example. If this value is null, the adapter does currently not belong to any network
`wPANID`	`unsigned short &` (out)	Reference to a 16-bit unsigned integer, which receives the short PAN-ID of the ZigBee network, which is used for addressing at the MAC level
`wShortAddress`	`unsigned short &` (out)	Reference to a 16-bit unsigned integer, which receives the network short address of the adapter on the ZigBee network. If this address is equal to 0xFFFF, the adapter is currently not operating on a network, i.e. it was not started yet.
`nChannel`	`unsigned char &` (out)	Reference to an 8-bit unsigned integer, which receives the current operating channel of the ZigBee adapter.
`bAssociationPermit`	`bool &` (out)	Reference to a Boolean, which receives the current permit joining (MAC association permit) state.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

```
ENSURE(SUCCEEDED(u7bfx::GetNetworkParameters(hDevice, qwExtendedPANID,
      wPANID, wShortAddress, nChannel, bAssociationPermit)));

std::wcout << L"Extended PANID: " << std::hex << std::setw(16)
      << std::setfill(L'0') << qwExtendedPANID << std::endl;

std::wcout << L"Channel: #" << std::dec << nChannel;

std::wcout << L", PANID: " << std::hex << std::setw(4) << std::setfill(L'0') << wPANID;

std::wcout << L", network address: " << std::hex << std::setw(4)
      << std::setfill(L'0') << wShortAddress << std::endl;
```

5.1.4.9. `SetDesignatedRole()` – Configure the ZigBee Device Role

Declaration

```
HRESULT STDAPICALLTYPE SetDesignatedRole(const HANDLE hDevice,
      const unsigned int nDeviceRole,
      const unsigned long long qwExtendedPANID = 0ull,
      const unsigned short wPANID = 0xffff,
      const unsigned short wShortAddress = 0xffff,
      const unsigned long dwChannelMask = 0x07fff800,
      const unsigned long long qwTrustCenterAddress = 0ull,
      const unsigned char bStartupControl = -1);
```

Synopsis

The ZigBee/USB adapter can operate in different device roles, in particular as (a) ZigBee Coordinator and Trust Center, (b) ZigBee Router, or (c) ZigBee End-Device[6]. Depending on device role and the start-up control parameter, some values of the ZigBee start-up attribute set (SAS) can be specified by the application. For example, the EPID to use when forming a network or joining a specific network, rejoining as a specific device etc.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nDeviceRole`	`unsigned int` (in)	Any of the following device roles: 0x00 – ZigBee Coordinator & Trust Center 0x01 – ZigBee Router 0x02 – ZigBee End-Device, non-sleeping 0x82 – ZigBee End-Device, sleeping
`qwExtendedPANID`	`unsigned long long` (in)	The 64-bit EPID of the network to join or form. If 0, the stack will use the adapter's 64-bit IEEE extended address as EPID. Applications may also provide a random value to create new networks with each new formation request.
`wPANID`	`unsigned short` (in)	16-bit PAN-ID of the network to join or form.
`wShortAddress`	`unsigned short` (in)	16-bit network short address on the network
`dwChannelMask`	`unsigned int` (in)	32-bit channel mask according to IEEE 802.15.4. Each bit corresponds to the according channel in the 2.4 GHz band, e.g. 0x00000800 = channel #11 … 0x04000000 = channel #26. For all 16 channels use 0x07FFF800
`qwTrustCenterAddress`	`unsigned long long` (in)	IEEE extended address of the Trust Center. Set to 0 to form a centralized network, set to 0xFFFFFFFFFFFFFFFF to form a distributed security network (ZigBee 3.0 and ZigBee Light Link only)
`bStartupControl`	`unsigned char` (in)	Controls start-up behaviour: 0 – Resume operation 1 – Form network 2 – Re-join network 3 – Join network

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

[6] Operation as sleeping and non-sleeping ZigBee End-Device is primarily for debugging purposes. Customer applications should operate the adapter either as ZigBee Coordinator and Trust Center or as ZigBee Router.

5.1.4.10. `Startup()` – Start the ZigBee Stack

Declaration

```
HRESULT STDAPICALLTYPE Startup(const HANDLE hDevice, const unsigned char nFlags,
      unsigned char &nStatus);
```

Synopsis

The controlling host application must explicitly start the Compact7B™ ZigBee stack embedded into the ZigBee/USB adapter by invoking this function. The stack provides fine-grained control over how the ZigBee stack starts and which actions it is allowed to perform:

1) Resume network operation (also called silent rejoin)
2) Perform a secure network rejoin
3) Perform a Trust Center rejoin (formerly known as “insecure” rejoin)
4) Associate from scratch
5) Form a new network

Instead of specifying allowed actions with this level of detail, the application can always use the default value of “0”, which results in the ZigBee adapter deriving the allowed actions from the current settings in the ZigBee Start-up Attribute Set (SAS).

The stack will attempt all allowed actions in the order listed above, i.e. first it will try to resume; if that fails, it will try a secure re-join; if that fails, it will attempt a Trust Center re-join; if that fails, it will try to associate to a new network from scratch; if that fails, it will try to form a new network.

When not using the default value, these operations must match with the start-up mode specified in the SAS, as configured with `SetDesignatedRole()` or preconfigured at the factory. The factory-fresh role for U1 is ZigBee Coordinator and Trust Center, and the initial start-up control value “1” instructs the stack to form a new network. Once successful, a start-up control value of “0” reflects the fact that the device is in a commissioned state and shall not form a network on subsequent invocations of the start-up procedure.

If a network start-up is already in progress, the call will return with a status code of `NWK:INVALID_REQUEST`. Otherwise, the immediate status will be `SUCCESS` (0), and the framework notifies the application via its start-up completion handler about the result of the start request.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nFlags`	`unsigned char` (in)	Either zero, to let the stack automatically determine the appropriate set of actions, or a combination of the following bit flags in descending priority: Bit #0 – Allow resume Bit #1 – Allow secure rejoin Bit #2 – Allow trust center rejoin Bit #3 – Allow association Bit #4 – Allow network formation
`nStatus`	`unsigned char &` (out)	An immediate status code for the attempt to start the network.

For convenience, u7bfx.h defines following enumeration values for use as start-up flags:

```
enum { startupAllowResume = 0x01, startupAllowSecureRejoin = 0x02,
       startupAllowTrustCenterRejoin = 0x04, startupAllowAssociation = 0x08,
       startupAllowNetworkFormation = 0x10, startupDefault = 0,
       startupAllowJoining = startupAllowResume | startupAllowSecureRejoin |
       startupAllowTrustCenterRejoin | startupAllowAssociation,
       startupAllowRejoining = startupAllowSecureRejoin |
       startupAllowTrustCenterRejoin, startupModes = 0x1f };
```

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.4.11. `SetOnNotifyStartupComplete()` – Install Start-up Completion Handler

Declaration

```
HRESULT STDAPICALLTYPE SetOnNotifyStartupComplete(const HANDLE hDevice,
      const ONNOTIFYSTARTUPCOMPLETEHANDLER pfnOnNotifyStartupComplete);
```

Synopsis

Installs a completion handler for the start-up procedure of the ZigBee networking stack. If `Startup()` succeeded (as determined by, both, the `HRESULT` return value and the `nStatus` code), the framework will call this user-defined handler when the start-up procedure actually completed and the ZigBee adapter is ready to be used for sending and receiving ZigBee traffic.

CAUTION: This callback is potentially invoked from another thread than the thread originally invoking `Startup()`. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnNotifyStartupComplete`	`ONNOTIFYSTARTUPCOMPLETEHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has the completed ZigBee networking stack start-up sequence.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONNOTIFYSTARTUPCOMPLETEHANDLER)
      (const HANDLE hDevice, unsigned char bStatus);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`bStatus`	`unsigned char` (in)	Result of the network start-up request, e.g. `SUCCESS`, `STARTUP_FAILURE`, etc.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.4.12. `SetCurrentChannel()` – Set the Current Channel

Declaration

```
HRESULT STDAPICALLTYPE SetCurrentChannel(const HANDLE hDevice,
      const unsigned char nChannel, unsigned char &nStatus);
```

Synopsis

Changes the current channel. This is only intended for Stub APS messaging, where the ZigBee/USB adapter is required to temporarily change channels, for example to perform a touch-link scan. After changing channels, always revert to the original ZigBee network channel[7].

The following IEEE 802.15.4 PHY status codes are available for convenience:

```
namespace phy
{
      enum { statusTransceiverOff = 8, statusReceiverEnabled = 6,
            statusTransmitterEnabled = 9, statusTransmitting = 2,
            statusReceiving = 1, statusSuccess = 7, statusChannelBusy = 0,
            statusChannelIdle = 4 };
}
```

Important Notice: Contrary to the usual habit of zero meaning `SUCCESS`, the IEEE 802.15.4 PHY defines `SUCCESS` as 7. For this API, this does only apply to `SetCurrentChannel()`. Do not confuse with `SUCCESS` (0) in other parts of this documentation and your code!

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nChannel`	`unsigned char` (in)	Number of the channel to change to.
`nStatus`	`unsigned char &` (out)	Reference to an 8-bit unsigned integer, which receives the IEEE 802.15.4 PHY status code for the channel change request.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

[7] This information is available via `GetNetworkParameters()`.

5.1.5. ZigBee Application Support Sublayer (APS) Interface

5.1.5.1. `ConfigureEndpoint()` – Allocate/Configure a ZigBee Endpoint

Declaration

```
HRESULT STDAPICALLTYPE ConfigureEndpoint(const HANDLE hDevice,
      const void *const pSimpleDescriptor, const size_t cbSimpleDescriptor,
      unsigned char &nStatus);
```

Synopsis

Allocates and configures a ZigBee endpoint on the adapter, such that the application can use it for sending and receiving ZigBee APS traffic.

Notice endpoint #0 is reserved for the ZDO, and endpoint #255 is reserved as the "broadcast to all endpoints" address. Specifying these endpoints in the simple descriptor will result in a `PARAMETER_INVALID_VALUE` status.

Trying to configure an endpoint, which is occupied by the adapter firmware, will result in a `NOT_ALLOWED` status.

The special endpoint #242 pertains to the ZigBee Green Power feature. By default, the adapter provides a Green Power Proxy on this endpoint, which a host application can promote to a Green Power Combined device, when it provides suitable Green Power sink functionality.

Only if the status code is `SUCCESS`, the application may use this endpoint for APS data transfers.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pSimpleDescriptor`	`void *` (in)	Pointer to a simple descriptor, in ZigBee over-the-air format, which the adapter shall apply to the endpoint.
`cbSimpleDescriptor`	`size_t` (in)	Size of the supplied simple descriptor, in bytes.
`nStatus`	`unsigned char &` (out)	Returns the status of the attempt to allocate or reconfigure an endpoint.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.5.2. `ClearEndpoint()` – Release a ZigBee Endpoint

Declaration

```
HRESULT STDAPICALLTYPE ClearEndpoint(const HANDLE hDevice,
      const unsigned char nEndpoint, unsigned char &nStatus);
```

Synopsis

Releases a previously allocated and configured ZigBee endpoint on the adapter. All inbound APS traffic for this endpoint will seize. Trying to clear an endpoint, which is under control of the adapter firmware, will result in a `NOT_ALLOWED` status.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nEndpoint`	`unsigned char` (in)	Identifies the endpoint, which shall no longer be active.
`nStatus`	`unsigned char &` (out)	Returns the status of the attempt to release an endpoint.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.5.3. `RequestData()` – Send APS Data Message

Declaration

```
HRESULT STDAPICALLTYPE RequestData(const HANDLE hDevice,
      const unsigned int dwRequestID,
      const unsigned char bDestinationAddressingMode,
      const unsigned short wDestinationAddress,
      const unsigned long long qwDestinationAddress,
      const unsigned char bDestinationEndpoint,
      const unsigned short wProfileID, const unsigned short wClusterID,
      const unsigned char bSourceEndpoint,
      const void *const pASDU, const unsigned short cbASDU,
      const unsigned char bTransmitOptions, const unsigned char bRadius,
      const unsigned char bEnhancedOptions);
```

Synopsis

Sends an APS datagram over the ZigBee network to a single device or a group of devices. A single device can be addressed using its network short address and a destination endpoint address, or using its IEEE extended address and destination endpoint address.

The source endpoint must have been configured via `ConfigureEndpoint()`.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwRequestID`	`unsigned int` (in)	A 32-bit identifier provided by the caller, which the framework will subsequently pass to the APS data confirmation handler to refer to this APS transaction.
`bDestinationAddressingMode`	`unsigned char` (in)	The destination addressing mode for this APS datagram: 1 = group-cast (and group address in `wDestinationAddress`), 2 = unicast via network short address (in `wDestinationAddress` and `bDestinationEndpoint`), 3 = unicast via IEEE extended address (in `qwDestinationAddress` and `bDestinationEndpoint`).
`wDestinationAddress`	`unsigned short` (in)	Network short address of the destination (in case of a unicast) or group address of the destination (in case of a group cast)
`qwDestinationAddress`	`unsigned long long` (in)	IEEE extended address of the destination (in case of a unicast)
`bDestinationEndpoint`	`unsigned char` (in)	Application endpoint on the destination (in case of a unicast).
`wProfileID`	`unsigned short` (in)	Application profile identifier, e.g. 0x0104 for ZigBee Home Automation, ZigBee Light Link and ZigBee 3.0
`wClusterID`	`unsigned short` (in)	Application cluster identifier, e.g. 0x0006 for the standard on/off cluster
`bSourceEndpoint`	`unsigned char` (in)	Application endpoint to use as source address. An application must exist on this endpoint and the endpoint must have previously been initialized by `ConfigureEndpoint()`.
`pASDU`	`const void *` (in)	Pointer to APS payload of `cbASDU` bytes length.

cbASDU	unsigned short (in)	Size of the APS payload provided in the memory location pointed to by pASDU.
bTransmitOptions	unsigned char (in)	APS transmit options. Please refer to APSDE-DATA.request for details.
bRadius	unsigned char (in)	Hop limit (time to live) for the message. Set to zero for the stack default value (currently 30), or specify another, typically smaller, hop limit.
bEnhancedOptions	unsigned char (in)	Enhanced transmit options. For example, bit #6 (0x40) enables broadcast reflection, an ubisys-specific APS feature that forwards a broadcast to the device itself for local processing in addition to sending the broadcast over-the-air. Refer to Compact7B™ documentation for full details.

Return Value

A HRESULT conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros SUCCEEDED() and FAILED() to evaluate the return value. Typically, the return code will be S_OK. In case of success, a confirmation handler will be invoked after APS transmission has completed to convey the result of the APS data transmission.

5.1.5.4. `SetOnConfirmData()` – Install Confirmation Handler for Outbound APS Data

Declaration

```
HRESULT STDAPICALLTYPE SetOnConfirmData(const HANDLE hDevice,
      const ONCONFIRMDATAHANDLER pfnOnConfirmData);
```

Synopsis

Installs a data indication handler for outbound APS datagrams sent using `RequestData()` over one of the endpoints the application has previously configured.

CAUTION: This callback is potentially invoked from another thread than the thread originally invoking `RequestData()`. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnConfirmData`	`ONCONFIRMDATAHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has completed transmission of an APS datagram.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONCONFIRMDATAHANDLER)(const HANDLE hDevice,
      unsigned int dwRequestID, unsigned int dwTimestamp,
      unsigned char bStatus);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwRequestID`	`unsigned int` (in)	The 32-bit request identifier originally supplied to `RequestData()`, when the application queued an APS data transmission request.
`dwTimestamp`	`unsigned long long` (in)	Time-stamp of the frame as reported by the IEEE 802.15.4 MAC layer.
`bStatus`	`unsigned char` (in)	If transmission was successful, this status code equals `SUCCESS` (0); else, it conveys an APS, NWK, or MAC status code identifying the reason of the failure.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.5.5. `SetOnIndicateData()` – Install Indication Handler for Inbound APS Data

Declaration

```
HRESULT STDAPICALLTYPE SetOnIndicateData(const HANDLE hDevice,
      const ONINDICATEDATAHANDLER pfnOnIndicateData);
```

Synopsis

Installs a data indication handler for inbound APS datagrams, which target one of the endpoints the application has previously configured.

CAUTION: This callback is potentially invoked from another thread than the application's main thread. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnIndicateData`	`ONINDICATEDATAHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it receives an APS datagram for the application.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONINDICATEDATAHANDLER)(const HANDLE hDevice,
      unsigned char bApplication, unsigned long long qwSourceAddress,
      unsigned short wSourceAddress, unsigned char bSourceEndpoint,
      unsigned char bSourceAddressingMode, unsigned short wDestinationAddress,
      unsigned char bDestinationEndpoint,
      unsigned char bDestinationAddressingMode, unsigned short wProfileID,
      unsigned short wClusterID, unsigned int dwTimestamp,
      unsigned char bLinkQuality, unsigned char bStatus,
      unsigned char bSecurityStatus, const void *const pASDU,
      const unsigned short cbASDU);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`bApplication`	`unsigned char` (in)	The application endpoint this APS datagram is for. For example, if the destination endpoint address is the broadcast endpoint (255) or this was a multicast, the framework provides multiple indications of the same message for each matching endpoint.
`qwSourceAddress`	`unsigned long long` (in)	IEEE extended address of the device originating the APS data frame.
`wSourceAddress`	`unsigned short` (in)	Network short address of the device originating the APS data frame.
`bSourceEndpoint`	`unsigned char` (in)	Application source endpoint on the device originating the APS data frame.

bSourceAddressingMode	unsigned char (in)	Source addressing mode, either 2 = network short address (in wDestinationAddress and bDestinationEndpoint), or 3 = IEEE extended address (in qwDestinationAddress and bDestinationEndpoint).
wDestinationAddress	unsigned short (in)	Either a group or network short address, as determined by bDestinationAddressingMode.
bDestinationEndpoint	unsigned char (in)	Destination endpoint as specified in the APS frame.
bDestinationAddressingMode	unsigned char (in)	Destination addressing mode, either 1 = group-cast (and group address in wDestinationAddress), or 2 = unicast via network short address (in wDestinationAddress and bDestinationEndpoint),
wProfileID	unsigned short (in)	Application profile identifier, e.g. 0x0104 for ZigBee Home Automation, ZigBee Light Link and ZigBee 3.0
wClusterID	unsigned short (in)	Application cluster identifier, e.g. 0x0006 for the standard on/off cluster
dwTimestamp	unsigned int (in)	Time-stamp of the frame as reported by the IEEE 802.15.4 MAC layer
bLinkQuality	unsigned char (in)	LQI of the frame as reported by the IEEE 802.15.4 MAC layer
bStatus	unsigned char (in)	This is currently always SUCCESS (0).
bSecurityUsed	bool (in)	True, if the frame was secured at the APS layer.
pASDU	const void * (in)	Pointer to a location in memory, which holds the APS payload conveyed in the datagram, the size of which is provided in cbASDU.
cbASDU	unsigned short (in)	Size of the APS payload at pASDU.

Return Value

A HRESULT conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros SUCCEEDED() and FAILED() to evaluate the return value. Typically, the return code will be S_OK.

5.1.5.6. `RequestDataStub()` – Send Stub APS Data Message

Declaration

```
HRESULT STDAPICALLTYPE RequestDataStub(const HANDLE hDevice,
      const unsigned int dwRequestID,
      const unsigned char bDestinationAddressingMode,
      const unsigned short wDestinationPANID,
      const unsigned short wDestinationAddress,
      const unsigned long long qwDestinationAddress,
      const unsigned short wProfileID, const unsigned short wClusterID,
      const void *const pASDU, const unsigned short cbASDU);
```

Synopsis

Sends datagram over the ZigBee Stub APS, a thin layer on top of the IEEE 802.15.4 MAC, which effectively bypasses the ZigBee networking stack. This feature is also known as inter-PAN messaging[8] and is the basis for touch-link commissioning as defined by the ZigBee Light Link application profile.

You might want to temporarily change channels with `SetCurrentChannel()` prior to calling `RequestDataStub()`.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwRequestID`	`unsigned int` (in)	A 32-bit identifier provided by the caller, which the framework will subsequently pass to the APS data confirmation handler to refer to this APS transaction.
`bDestinationAddressingMode`	`unsigned char` (in)	The destination addressing mode for this APS datagram: 0 = group-cast or unicast to all matching targets specified in the binding table, 1 = group-cast (and group address in `wDestinationAddress`), 2 = unicast via network short address (in `wDestinationAddress` and `bDestinationEndpoint`), 3 = unicast via IEEE extended address (in `qwDestinationAddress` and `bDestinationEndpoint`).
`wDestinationPANID`	`unsigned short` (in)	Network identifier of the destination.
`wDestinationAddress`	`unsigned short` (in)	Network short address of the destination (in case of a unicast) or group address of the destination (in case of a group cast)
`qwDestinationAddress`	`unsigned long long` (in)	IEEE extended address of the destination (in case of a unicast)
`wProfileID`	`unsigned short` (in)	Application profile identifier, e.g. 0xC05E for ZigBee Light Link
`wClusterID`	`unsigned short` (in)	Application cluster identifier, e.g. 0x1000 for the touch-link commissioning cluster
`pASDU`	`const void *` (in)	Pointer to APS payload of `cbASDU` bytes length.
`cbASDU`	`unsigned short` (in)	Size of the APS payload provided in the memory location pointed to by `pASDU`.

[8] The stub APS was originally designed for ZigBee Smart Energy to enable low-cost in-home price displays

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`. In case of success, a confirmation handler will be invoked after transmission has completed to convey the result of the APS stub data transmission.

5.1.5.7. `SetOnConfirmDataStub()` – Install Confirmation Handler Stub APS Data

Declaration

```
HRESULT STDAPICALLTYPE SetOnConfirmDataStub(const HANDLE hDevice,
      const ONCONFIRMDATASTUBHANDLER pfnOnConfirmData);
```

Synopsis

Installs a data indication handler for outbound APS datagrams sent using `RequestData()` over one of the endpoints the application has previously configured.

CAUTION: This callback is potentially invoked from another thread than the thread originally invoking `RequestData()`. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnConfirmData`	`ONCONFIRMDATASTUBHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has completed transmission of a Stub APS datagram.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONCONFIRMDATASTUBHANDLER)(const HANDLE hDevice,
      unsigned int dwRequestID, unsigned int dwTimestamp,
      unsigned char bStatus);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwRequestID`	`unsigned int` (in)	The 32-bit request identifier originally supplied to `RequestDataStub()`, when the application queued a Stub APS data transmission request.
`dwTimestamp`	`unsigned long long` (in)	Time-stamp of the frame as reported by the IEEE 802.15.4 MAC layer.
`bStatus`	`unsigned char` (in)	If transmission was successful, this status code equals `SUCCESS` (0); else, it conveys an APS, NWK, or MAC status code identifying the reason of the failure.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.5.8. `SetOnIndicateDataStub()` – Install Indication Handler for Stub APS Data

Declaration

```
HRESULT STDAPICALLTYPE SetOnIndicateDataStub(const HANDLE hDevice,
      const ONINDICATEDATASTUBHANDLER pfnOnIndicateData);
```

Synopsis

Installs a data indication handler for inbound Stub APS datagrams.

CAUTION: This callback is potentially invoked from another thread than the application's main thread. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnIndicateData`	`ONINDICATEDATASTUBHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it receives a Stub APS datagram for the application.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONINDICATEDATASTUBHANDLER)
      (const HANDLE hDevice, unsigned long long qwSourceAddress,
      unsigned short wSourceAddress, unsigned short wSourcePANID,
      unsigned char bSourceAddressingMode,
      unsigned long long qwDestinationAddress, unsigned short wDestinationAddress,
      unsigned short wDestinationPANID, unsigned char bDestinationAddressingMode,
      unsigned int dwTimestamp, unsigned char bLinkQuality,
      const unsigned short wProfileID, const unsigned short wClusterID,
      const void *const pASDU, const unsigned short cbASDU);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`qwSourceAddress`	`unsigned long long` (in)	IEEE extended address of the device originating the APS data frame.
`wSourceAddress`	`unsigned short` (in)	Network short address of the device originating the APS data frame.
`wSourcePANID`	`unsigned short` (in)	Network identifier of the device originating the Stub APS data frame.
`bSourceAddressingMode`	`unsigned char` (in)	Source addressing mode, either 2 = network short address (in `wDestinationAddress` and `bDestinationEndpoint`), or 3 = IEEE extended address (in `qwDestinationAddress` and `bDestinationEndpoint`).
`qwDestinationAddress`	`unsigned long long` (in)	IEEE extended address of the device targeted by the Stub APS data frame.
`wDestinationAddress`	`unsigned short` (in)	Network short address of the device targeted by the APS data frame.
`wDestinationPANID`	`unsigned short` (in)	Network identifier of the device targeted by the Stub APS data frame.

bDestinationAddressingMode	unsigned char (in)	Source addressing mode, either 1 = group-cast (and group address in wDestinationAddress), 2 = unicast via network short address (in wDestinationAddress and bDestinationEndpoint),
dwTimestamp	unsigned int (in)	Time-stamp of the frame as reported by the IEEE 802.15.4 MAC layer
bLinkQuality	unsigned char (in)	LQI of the frame as reported by the IEEE 802.15.4 MAC layer
wProfileID	unsigned short (in)	Application profile identifier, e.g. 0xC05E for ZigBee Light Link
wClusterID	unsigned short (in)	Application cluster identifier, e.g. 0x1000 for the touch-link commissioning cluster
pASDU	const void * (in)	Pointer to a location in memory, which holds the Stub APS payload conveyed in the datagram, the size of which is provided in cbASDU.
cbASDU	unsigned short (in)	Size of the Stub APS payload at pASDU.

Return Value

A HRESULT conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros SUCCEEDED() and FAILED() to evaluate the return value. Typically, the return code will be S_OK.

5.1.6. ZigBee Device Object (ZDO) Interface

5.1.6.1. `RequestDeviceTransaction()` – Perform ZDO Transaction

Declaration

```
HRESULT STDAPICALLTYPE RequestDeviceTransaction(const HANDLE hDevice,
      const unsigned int dwRequestID,
      const unsigned char bDestinationAddressingMode,
      const unsigned short wDestinationAddress,
      const unsigned long long qwDestinationAddress,
      const unsigned short wClusterID, const void *const pTransactionData,
      const unsigned short cbTransactionData,
      const unsigned char bTransmitOptions, const unsigned char bRadius,
      const unsigned char bEnhancedOptions, const unsigned int dwTimeout);
```

Synopsis

Sends a ZDO datagram over the ZigBee network and optionally waits for a response.

Notice that the Compact7B™ stack embedded into the adapter's firmware is also using the ZDO internally, which is why the adapter maintains ZDP transaction sequence counters; the ZDO is a shared resource. In addition, it will always perform necessary processing of incoming ZDO frames, like device_annce and additionally notify the application where appropriate. The host application need not (and cannot) implement any ZDO behaviour. This is contrary to the APS data service, where the application has exclusive access and ownership.

The APS broadcast reflection feature is useful for permit joining operations, such that the adapter will both permit joining locally and broadcast the permit joining frame to other routers in the network.

Currently, only the first response will be taken into account, should there be multiple responses to a single request.

The ZDO uses stock endpoint #0 and profile ID 0x0000, ZigBee Device Profile (ZDP). There is no need to configure an endpoint via `ConfigureEndpoint()` for executing ZDO transactions.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwRequestID`	`unsigned int` (in)	A 32-bit identifier provided by the caller, which the framework will subsequently pass to the APS data confirmation handler to refer to this APS transaction.
`bDestinationAddressingMode`	`unsigned char` (in)	The destination addressing mode for this APS datagram: 1 = group-cast (and group address in `wDestinationAddress`), 2 = unicast via network short address (in `wDestinationAddress` and `bDestinationEndpoint`), 3 = unicast via IEEE extended address (in `qwDestinationAddress` and `bDestinationEndpoint`).

wDestinationAddress	unsigned short (in)	Network short address of the destination (in case of a unicast) or group address of the destination (in case of a group cast)
qwDestinationAddress	unsigned long long (in)	IEEE extended address of the destination (in case of a unicast)
wClusterID	unsigned short (in)	ZDP cluster identifier, e.g. 0x0036 for a mgmt_permit_joining_req
pTransactionData	const void * (in)	Pointer to ZDO transaction data of cbTransactionData bytes length. This is only the transaction data part of a ZDP frame, without the transaction sequence number.
cbTransactionData	unsigned short (in)	Size of the ZDO transaction data in the memory location pointed to by pTransactionData.
bTransmitOptions	unsigned char (in)	APS transmit options. Please refer to APSDE-DATA.request for details.
bRadius	unsigned char (in)	Hop limit (time to live) for the message. Set to zero for the stack default value (currently 30), or specify another, typically smaller, hop limit.
bEnhancedOptions	unsigned char (in)	Enhanced transmit options. For example, bit #6 (0x40) enables broadcast reflection, an ubisys-specific APS feature that forwards a broadcast to the device itself for local processing in addition to sending the broadcast over-the-air. Refer to Compact7B™ documentation for full details.
dwTimeout	unsigned int (in)	Specifies the time, in microseconds, to wait for a response frame in order to complete the transaction. A value of 0 means do not wait for a response and should be used when the caller does not expect a response frame, e.g. when broadcasting a mgmt_permit_joining_req.

Return Value

A HRESULT conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros SUCCEEDED() and FAILED() to evaluate the return value. Typically, the return code will be S_OK. In case of success, a confirmation handler will be invoked after APS transmission has completed to convey the result of the APS data transmission.

5.1.7. Raw Packet Capture Interface

The ZigBee/USB Adapter U1 has the ability to acquire raw MAC frames[9]. With this feature, application developers can build protocol analysers, test and production line equipment using U1 for any IEEE 802.15.4 based protocol, including ZigBee, ZigBee Green Power, 6lowpan, Thread, WirelessHART ISA 100.11a, and others.

Important Notice: This feature is not to be confused with the ubisys IEEE 802.15.4 USB Stick for Wireshark, which is a different product that runs a different firmware (Network Adapter using the RNDIS USB protocol) and does not provide any ZigBee functionality.

5.1.7.1. `EnablePromiscuousMode()` – Set the Current Channel

Declaration

```
HRESULT STDAPICALLTYPE EnablePromiscuousMode(const HANDLE hDevice,
      unsigned char &nStatus, const bool bEnable = true,
      const unsigned char nChannel = 11, const bool bAutoStartStop = true,
      const bool bForceStartStop = false);
```

Synopsis

Enters or leaves IEEE 802.15.4 MAC promiscuous mode. In promiscuous mode, the ZigBee/USB adapter operates as a raw capture device (protocol sniffer), which acquires MAC frames and presents them unmodified to the host application without further processing. The host application must install a suitable handler via `SetOnIndicatePromiscuousData()`. It is possible to switch back and forth between normal operating mode as a node on a ZigBee network and raw packet acquisition mode.

Important Notice: While in MAC promiscuous mode, the ZigBee/USB adapter U1 is off the ZigBee network. It does not process incoming frames other than forwarding to the host application.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nStatus`	`unsigned char &` (out)	Reference to an 8-bit unsigned integer, which receives the IEEE 802.15.4 MAC status code for the mode change request.
`bEnable`	`bool` (in)	Enables or disables MAC promiscuous mode.
`nChannel`	`unsigned char` (in)	Number of the channel to start packet acquisition on.
`bAutoStartStop`	`bool` (in)	Automatically start/stop the capture, based on `bEnable`.
`bForceStartStop`	`bool` (in)	Force starting/stopping the capture.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

[9] Since application firmware version 1.70

5.1.7.2. `SetOnIndicatePromiscuousData()` – Install Raw MAC Capture Handler

Declaration

```
HRESULT STDAPICALLTYPE SetOnIndicatePromiscuousData(const HANDLE hDevice,
      const ONINDICATEPROMISCUOUSDATAHANDLER pfnOnIndicatePromiscousData);
```

Synopsis

Installs a data indication handler for raw MAC frames.

CAUTION: This callback is potentially invoked from another thread than the application's main thread. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnIndicatePromiscousData`	`ONINDICATEPROMISCUOUSDATAHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it receives a MAC frame.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT(STDAPICALLTYPE *ONINDICATEPROMISCUOUSDATAHANDLER)
      (const HANDLE hDevice, unsigned int dwTimestamp,
      unsigned char bChannel, unsigned char bLinkQuality,
      signed char bRSSI, unsigned char bCorrelation,
      const void *const pMPDU, const unsigned short cbMPDU);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwTimestamp`	`unsigned int` (in)	Time-stamp of the frame as reported by the IEEE 802.15.4 MAC layer
`bChannel`	`unsigned char` (in)	Number of the IEEE 802.15.4 channel this frame was acquired on
`bLinkQuality`	`unsigned char` (in)	LQI of the frame as reported by the IEEE 802.15.4 MAC layer
`bRSSI`	`signed char` (in)	Received signal strength indicator of the frame as reported by the IEEE 802.15.4 radio
`bCorrelation`	`unsigned char` (in)	Raw receiver correlation value of the frame as reported by the IEEE 802.15.4 radio
`pMPDU`	`const void *` (in)	Pointer to a location in memory, which holds the MPDU, the size of which is provided in `cbMPDU`.
`cbMPDU`	`unsigned short` (in)	Size of the MPDU at `pMPDU`.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.7.3. `SetOnConfirmDeviceTransaction()` – Install ZDO Completion Handler

Declaration

```
HRESULT STDAPICALLTYPE SetOnConfirmDeviceTransaction(const HANDLE hDevice,
      const ONCONFIRMDEVICETRANSACTIONHANDLER pfnOnConfirmDeviceTransaction);
```

Synopsis

Installs a completion handler for ZDO transactions issued via `RequestDeviceTransaction()`. The framework will call this user-defined handler for each completed ZDO transaction to convey the results of the transaction.

CAUTION: This callback is potentially invoked from another thread than the thread originally invoking `RequestDeviceTransaction()`. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnConfirmDeviceTransaction`	`ONCONFIRMDEVICETRANSACTIONHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has completed a ZDO transaction.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONCONFIRMDEVICETRANSACTIONHANDLER)
      (const HANDLE hDevice, unsigned int dwRequestID, unsigned char bResult,
            unsigned long long qwSourceAddress, unsigned short wSourceAddress,
            unsigned char bSourceAddressingMode,
            unsigned short wDestinationAddress, unsigned short wClusterID,
            unsigned char bStatusTX, unsigned int dwTimestampTX,
            unsigned int dwTimestampRX, unsigned char bLinkQuality,
            unsigned char bStatusRX, bool bSecurityUsed,
            const void *const pResponse, const unsigned short cbResponse);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwRequestID`	`unsigned int` (in)	The 32-bit request identifier originally supplied to `RequestData()`, when the application queued an APS data transmission request.
`qwSourceAddress`	`unsigned long long` (in)	IEEE extended address of the device originating the APS data frame.
`wSourceAddress`	`unsigned short` (in)	Network short address of the device originating the APS data frame.
`bSourceAddressingMode`	`unsigned char` (in)	Source addressing mode, either 2 = network short address (in `wDestinationAddress` and `bDestinationEndpoint`), or 3 = IEEE extended address (in `qwDestinationAddress` and `bDestinationEndpoint`).

wDestinationAddress	unsigned short (in)	Destination addressing mode, either 1 = group-cast (and group address in wDestinationAddress), 2 = unicast via network short address (in wDestinationAddress and bDestinationEndpoint),
wClusterID	unsigned short (in)	Application cluster identifier, e.g. 0x0006 for the standard on/off cluster
bStatusTX	unsigned char (in)	Status of the transmission attempt, any APS, NWK, or MAC status code.
dwTimestampTX	unsigned int (in)	Time-stamp of the request frame as reported by the IEEE 802.15.4 MAC layer
dwTimestampRX	unsigned int (in)	Time-stamp of the response frame as reported by the IEEE 802.15.4 MAC layer
bLinkQuality	unsigned char (in)	LQI of the frame as reported by the IEEE 802.15.4 MAC layer
bStatusRX	unsigned char (in)	This is currently always SUCCESS (0).
bSecurityUsed	bool (in)	True, if the frame was secured at the APS layer.
pResponse	const void * (in)	Pointer to a location in memory, which holds the ZDO response, the size of which is specified in cbResponse.
cbResponse	unsigned short (in)	Size of the ZDO response at pResponse.

Return Value

A HRESULT conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros SUCCEEDED() and FAILED() to evaluate the return value. Typically, the return code will be S_OK.

5.1.7.4. `SetOnNotifyDeviceAnnouncement()` – Install ZDO Announcement Handler

Declaration

```
HRESULT STDAPICALLTYPE SetOnNotifyDeviceAnnouncement(const HANDLE hDevice,
      const ONNOTIFYDEVICEANNOUNCEMENTHANDLER pfnOnNotifyDeviceAnnoucement);
```

Synopsis

Installs a notification handler for ZDO device announcements received from devices joining the network for the first time or re-joining the network after temporarily losing network connectivity, potentially after power-up, etc.

This is usually a good time to configure devices joining the network for the first time.

Notice: You will also receive device announcements for ZigBee Green Power devices. Such announcements have an IEEE extended address marked as invalid (`FF:FF:FF:FF:FF:FF:FF:FF`), and should typically be ignored.

CAUTION: This callback is potentially invoked from another thread than the main thread. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnNotifyDeviceAnnoucement`	`ONNOTIFYDEVICEANNOUNCEMENTHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has received a ZDO device announcement.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONNOTIFYDEVICEANNOUNCEMENTHANDLER)
      (const HANDLE hDevice, const unsigned char bApplication,
      unsigned long long qwAddress, unsigned short wAddress,
      unsigned char bCapabilities, unsigned int dwTimestamp);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`bApplication`	`unsigned char` (in)	The application endpoint this notification is for. Each registered application endpoint will receive a notification.
`qwAddress`	`unsigned long long` (in)	IEEE extended address of the device joining or re-joining the network.
`wAddress`	`unsigned short` (in)	Network short address of the device joining or re-joining the network.
`bCapabilities`	`unsigned char` (in)	The IEEE 802.15.4 MAC layer capabilities of the device joining or re-joining the network.
`dwTimestamp`	`unsigned int` (in)	Time-stamp of the device announcement frame as reported by the IEEE 802.15.4 MAC layer

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.7.5. `SetOnNotifyPermitJoining()` – Install Permit Joining Handler

Declaration

```
HRESULT STDAPICALLTYPE SetOnNotifyPermitJoining(const HANDLE hDevice,
      const ONNOTIFYPERMITJOININGHANDLER pfnOnNotifyPermitJoining);
```

Synopsis

Installs a notification handler for changes to the adapter's permit joining state. This covers both cases, the ZigBee adapter opening the network for joining, or another device on the network opening the network for joining. It is not directly connected to the ZDO, rather to the MAC layer.

CAUTION: This callback is potentially invoked from another thread than the main thread. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`pfnOnNotifyPermitJoining`	`ONNOTIFYPERMITJOININGHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when the local permit joining state changes.

Callback Signature

The callback handler must have the following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONNOTIFYPERMITJOININGHANDLER)
      (const HANDLE hDevice, bool bAssociationPermit);
```

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`bAssociationPermit`	`bool` (in)	True if the IEEE 802.15.4 MAC layer currently permits association of new devices, false otherwise.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.8. Utility Functions

5.1.8.1. `GetStatusString()` – ZigBee status code as text

Declaration

```
HRESULT STDAPICALLTYPE GetStatusString(const LPTSTR pszStatus,
       const size_t cwStatus, const unsigned char nStatus);
```

Synopsis

Returns a human-readable string for the status code passed-in by the caller. Covers status codes of the IEEE 802.15.4 Medium Access Control Layer (MAC), ZigBee Network Layer (NWK) and ZigBee Application Support Sublayer (APS). The string is suitable for creating debug output messages. It is typically not useful for presentation to end-users, as it is extremely technical and English only. Examples are “SUCCESS” for a status of zero, “NWK:INVALID_PARAMETER” for status code 0xC1, etc. When possible, the function will denote the subsystem where the status code originated as either “MAC”, “NWK”, or “APS”. For unknown status codes, it will provide a hexadecimal number as ASCII string.

Arguments

Argument	Type (Direction)	Description
`pszStatus`	`LPTSTR` (in for location, out for data stored at this location)	Pointer to a string buffer, which will receive a status string with a textual representation for the status code supplied in `nStatus`. The buffer must be large enough to store `cwStatus` characters, including a trailing null terminator.
`cwStatus`	`size_t` (in)	Number of characters, including a trailing zero, which can be stored in the output buffer. Notice this is the number of characters, not the size in bytes.
`nStatus`	`unsigned char` (in)	A status code from the IEEE 802.15.4 MAC, the ZigBee Network Layer (NWK), or the ZigBee Application Support Sublayer (APS).

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

```
std::wstring GetStatusString(const unsigned char nStatus)
{
       wchar_t szStatus[32];

       VERIFY(SUCCEEDED(u7bfx::GetStatusString(szStatus, _countof(szStatus),
              nStatus)));

       return std::wstring(szStatus);
}


HRESULT STDAPICALLTYPE OnConfirmData(const HANDLE hDevice,
       unsigned int dwRequestID, unsigned int dwTimestamp, unsigned char bStatus)
{
       std::wcout << L"APSDE-DATA.confirm (request #" << std::dec << dwRequestID
              << L", status = " << std::hex << std::setw(2) << std::setfill(L'0')
              << static_cast<unsigned int>(bStatus) << L" - "
              << GetStatusString(bStatus).c_str() << L", time-stamp = " << std::dec
              << dwTimestamp << L")" << std::endl;

       // Handle the APSDE-DATA.confirm...
```

```
}
```

5.1.8.2. `GetDeviceObjectStatusString()` – ZDO status code as text

Declaration

```
HRESULT STDAPICALLTYPE GetDeviceObjectsStatusString(const LPTSTR pszStatus,
      const size_t cwStatus, const unsigned char nStatus);
```

Synopsis

Returns a human-readable string for the status code passed-in by the caller. Covers status codes of the ZigBee Device Objects (ZDO) and for unknown status codes, `GetStatusString()` will be called implicitly, adding coverage for MAC, NWK and APS. The string is suitable for creating debug output messages. It is typically not useful for presentation to end-users, as it is extremely technical and English only. Examples are "SUCCESS" for a status of zero, "ZDO:INV_REQUESTTYPE" for status code 0x80, etc.

Arguments

Argument	Type (Direction)	Description
`pszStatus`	`LPTSTR` (in for location, out for data stored at this location)	Pointer to a string buffer, which will receive a status string with a textual representation for the status code supplied in `nStatus`. The buffer must be large enough to store `cwStatus` characters, including a trailing null terminator.
`cwStatus`	`size_t` (in)	Number of characters, including a trailing zero, which can be stored in the output buffer. Notice this is the number of characters, not the size in bytes.
`nStatus`	`unsigned char` (in)	A status code from the IEEE 802.15.4 MAC, the ZigBee Network Layer (NWK), the ZigBee Application Support Sublayer (APS), or the ZigBee Device Object (ZDO).

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Refer to the example in section 0 for `GetStatusString()`, which is very similar.

5.1.8.3. `GetClusterLibraryStatusString()` – ZCL status code as text

Declaration

```
HRESULT STDAPICALLTYPE GetClusterLibraryStatusString(const LPTSTR pszStatus,
      const size_t cwStatus, const unsigned char nStatus);
```

Synopsis

Returns a human-readable string for the status code passed-in by the caller. Covers status codes of the ZigBee Cluster Library (ZCL). The string is suitable for creating debug output messages. It is typically not useful for presentation to end-users, as it is extremely technical and English only. Examples are “ZDO:SUCCESS” for a status of zero, “ZDO:FAILURE” for status code 0x01, etc. For unknown status codes, `GetStatusString()` will be called implicitly.

Arguments

Argument	Type (Direction)	Description
`pszStatus`	`LPTSTR` (in for location, out for data stored at this location)	Pointer to a string buffer, which will receive a status string with a textual representation for the status code supplied in `nStatus`. The buffer must be large enough to store `cwStatus` characters, including a trailing null terminator.
`cwStatus`	`size_t` (in)	Number of characters, including a trailing zero, which can be stored in the output buffer. Notice this is the number of characters, not the size in bytes.
`nStatus`	`unsigned char` (in)	A status code from the IEEE 802.15.4 MAC, the ZigBee Network Layer (NWK), the ZigBee Application Support Sublayer (APS), or the ZigBee Cluster Library (ZCL).

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

Example

Refer to the example in section 0 for `GetStatusString()`, which is very similar.

5.1.8.4. `GetDataStatistics()` – Obtain APS statistics

Declaration

```
HRESULT STDAPICALLTYPE GetDataStatistics(const HANDLE hDevice,
      unsigned int &nDataRequests, unsigned int &nDataConfirms,
      unsigned int &nDataIndications);
```

Synopsis

Returns the number of APS data requests, data confirmations and data indications exchanged over the adapter's APS interface. These are diver metrics, not metrics collected at the ZigBee stack level. Mainly for diagnostic purposes at early stages of integration into customer applications.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nDataRequests`	`unsigned int` (in)	Number of data requests received from the application.
`nDataConfirms`	`unsigned int` (in)	Number of data confirmations issued towards the application.
`nDataIndications`	`unsigned int` (in)	Number of data indications issued towards the application.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.8.5. `GetDeviceTransactionStatistics()` – Obtain ZDO statistics

Declaration

```
HRESULT STDAPICALLTYPE GetDeviceTransactionStatistics(const HANDLE hDevice,
      unsigned int &nRequests, unsigned int &nConfirms);
```

Synopsis

Returns the number of ZDO requests and confirmations exchanged over the adapter's ZDO interface. These are driver metrics, not metrics collected at the ZigBee stack level. Mainly for diagnostic purposes at early stages of integration into customer applications.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`nRequests`	`unsigned int` (in)	Number of ZDO requests received from the application.
`nConfirms`	`unsigned int` (in)	Number of ZDO confirmations issued towards the application.

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.8.6. `RequestNetworkDiscovery()` – Scan for ZigBee Networks in the Vicinity

Declaration

```
HRESULT STDAPICALLTYPE RequestNetworkDiscovery(const HANDLE hDevice,
       const unsigned int dwScanChannels, const unsigned char bScanDuration,
       const ONCONFIRMNETWORKDISCOVERYHANDLER pfnOnConfirmNetworkDiscovery);
```

Synopsis

Scans a set of channels for ZigBee and other IEEE 802.15.4 networks. When complete, the application-defined callback receives a list of network descriptors. With this feature, an application might display a site survey, and provide users with a choice of networks to join, for example.

CAUTION: The completion handler is potentially invoked from another thread than the thread originally invoking `RequestNetworkDiscovery()`. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwScanChannels`	`unsigned int` (in)	A 32-bit channel mask indicating which channels to scan. For example, 0x07FFF800 scans all 16 channels in the 2.4GHz band.
`bScanDuration`	`unsigned char` (in)	Duration of the active scan on each channel, as per IEEE 802.15.4. Refer to MLME-SCAN.request for details.
`pfnOnConfirmNetworkDiscovery`	`ONCONFIRMNETWORKDISCOVERYHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has completed the network scan.

The completion handler has following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONCONFIRMNETWORKDISCOVERYHANDLER)
       (const HANDLE hDevice, unsigned char bStatus,
       const unsigned int nNetworkCount,
       const CZigBeeNetworkDescriptor *const pDescriptors);
```

And each network will be conveyed as a CZigBeeNetworkDescriptor:

```
////////////////////////////////////////////////////////////////////////////
// CZigBeeNetworkDescriptor

class CZigBeeNetworkDescriptor
{
       // Attributes
       public:
              // The 64-bit PAN identifier of the network
              unsigned long long m_qwExtendedPANID;

              // The current logical channel occupied by the network
              unsigned char m_nLogicalChannel;

              // A ZigBee stack profile identifier indicating the stack profile in
```

```
            // use in the discovered network
            unsigned char m_nStackProfile;

            // The version of the ZigBee protocol in use in the discovered network
            unsigned char m_nZigBeeVersion;

            // This specifies how often the MAC sub-layer beacon is to be
            // transmitted by a given device on the network
            unsigned char m_nBeaconOrder;

            // For beacon-oriented networks, that is, beacon order < 15, this
            // specifies the length of the active period of the superframe
            unsigned char m_nSuperframeOrder;

            // A value of TRUE indicates that at least one ZigBee router on the
            // network currently permits joining, i.e. its NWK has been issued an
            // NLME-PERMIT-JOINING primitive and, the time limit if given, has not
            // yet expired
            bool m_bPermitJoining;

            // This value is set to true if the device is capable of accepting join
            // requests from router-capable devices
            bool m_bRouterCapacity;

            // This value is set to true if the device is capable of accepting join
            // requests from end devices and set to FALSE otherwise
            bool m_bEndDeviceCapacity;
};
```

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.1.8.7. `RequestEnergyDetectionScan()` – Measure Energy in Channels

Declaration

```
HRESULT STDAPICALLTYPE RequestEnergyDetectionScan(const HANDLE hDevice,
      const unsigned int dwScanChannels, const unsigned char bScanDuration,
      const ONCONFIRMENERGYDETECTIONSCANHANDLER pfnOnConfirmEnergyDetectionScan);
```

Synopsis

Measures the energy in a set of channels. This could be useful for determining background noise and interference levels for diagnostic purposes, or for selecting a new operating channel for the network. When complete, the application-defined callback receives a list of energy values, one byte for each scanned channel.

CAUTION: The completion handler is potentially invoked from another thread than the thread originally invoking `RequestEnergyDetectionScan()`. Make sure your handler is thread-safe.

Arguments

Argument	Type (Direction)	Description
`hDevice`	`HANDLE` (in)	A device handle previously returned by a successful call to `OpenDevice()`.
`dwScanChannels`	`unsigned int` (in)	A 32-bit channel mask indicating which channels to scan. For example, 0x07FFF800 scans all 16 channels in the 2.4GHz band.
`bScanDuration`	`unsigned char` (in)	Duration of the active scan on each channel, as per IEEE 802.15.4. Refer to MLME-SCAN.request for details.
`pfnOnConfirmEnergyDetectionScan`	`ONCONFIRMENERGYDETECTIONSCANHANDLER` (in)	Pointer to an application-defined callback, which the framework invokes when it has completed the energy scan.

The completion handler has following signature:

```
typedef HRESULT (STDAPICALLTYPE *ONCONFIRMENERGYDETECTIONSCANHANDLER)
      (const HANDLE hDevice, unsigned char bStatus,
      const unsigned int dwScannedChannels,
      const unsigned char *const pEnergyLevels);
```

Return Value

A `HRESULT` conveying the success/failure of the operation and potentially other information, like error codes in case of a failure. You should use the macros `SUCCEEDED()` and `FAILED()` to evaluate the return value. Typically, the return code will be `S_OK`.

5.2. 7Bfx™ for Linux

On Linux the 7Bfx™ API is provided as a static library, using UTF-8 for different architectures.

- Release, UTF-8: `lib7bfx.a`
- Debug, UTF-8: `lib7bfxd.a`

5.3. 7Bfx™ for Microsoft Windows

On Windows, the 7Bfx™ API is provided as a dynamic link library (DLL). Currently, ubisys provides Unicode UTF-16 builds of the DLLs for 32- and 64-bit systems in debug and release configurations. It runs on any 32-bit or 64-bit Windows, starting from Windows XP, currently including Windows 10:

- 32-bit Intel x86, Release, UTF-16: `7bfx.lib` and `7bfx.dll`
- 64-bit Intel x86, Release, UTF-16: `7bfx-x64.lib` and `7bfx-x64.dll`
- 32-bit Intel x86, Debug, UTF-16: `7bfxd.lib` and `7bfxd.dll`
- 64-bit Intel x86, Debug, UTF-16: `7bfxd-x64.lib` and `7bfxd-x64.dll`

The ubisys Network Manager™ software on Windows also uses this DLL to connect to U1 devices attached to a Windows computer.

You can use the DLL with any programming language and toolchain, which supports DLLs. However, ubisys currently only provides a C++ SDK. Therefore, you may have to provide the symbol definitions yourself for the language of your choosing. For example, when using C# you could use `DLLImport` annotations:

```
[DLLImport("7bfx-x64.dll", EntryPoint = "#1", CharSet = CharSet.Unicode,
SetLastError=false, CallingConvention=CallingConvention.StdCall)]
public static extern int Initialize(ref IntPtr handle);
```

For plain C, small tweaks of the u7bfx header would be required in order not to use C++ features like namespaces and references.

6.1. Hardware Installation

Plug the USB dongle into any available USB slot on your PC, which is able to supply 50mA of operating current. All USB ports (including passive hubs) should be suitable.

6.2. Software Installation

Currently, the ZigBee USB/Adapter is being used in conjunction with the “ubisys ZigBee Network Manager” (a Windows software application) and is also embedded into the ubisys ZigBee/Ethernet Gateway G1, which runs Linux. Device drivers for Windows, Linux and other operating systems are available upon request, as well as a complete documentation of the native ZigBee/USB protocol. Also available is a complete implementation of the ZigBee/IP Gateway Device with raw binary TCP/IP interface (GRIP), which runs on Linux, and is the core of the ZigBee/Ethernet Gateway ubisys G1.

Notice: The information mentioned above is only available to qualified customers for high-volume projects. Please contact ubisys sales for enquiries.

6.2.1. Linux udev Rules

Add a new udev rule provide read/write access to applications using the persistent USB rules (for example /etc/udev/rules.d/70-persistent-usb.rules in ubuntu), for instance:

```
# This file maintains persistent names for USB devices.
# See udev(7) for syntax.
#
# Entries are automatically added by the 75-cd-aliases-generator.rules
# file; however you are also free to add your own entries provided you
# add the ENV{GENERATED}=1 flag to your own rules as well.

#USB devices

SUBSYSTEM=="usb",ENV{DEVTYPE}=="usb_device",ATTRS{idVendor}=="19a6",ATTRS{idProduct}=="00
04",MODE="0666"
```

Then reload the rules and retrigger:

```
$ udevadm control --reload-rules
$ udevadm trigger
```

7. INITIAL DEVICE START-UP

The ZigBee/USB Adapter's start-up behaviour is completely defined by the application controlling it.

8. MAN-MACHINE INTERFACE (MMI)

The application must provide a suitable man-machine-interface (e.g. on a computer screen or using a LED and a push-button) for commissioning and factory reset. The ZigBee/USB Adapter U1 does not contain any MMI.

9. ZIGBEE INTERFACE

Please refer to the following IEEE and ZigBee Alliance documents, which apply to this product:

[R1] IEEE Standard 802 – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)
[R2] ZigBee Specification, Revision 21, Document No. 05-3474-21
[R3] ZigBee 2015 Layer PICS and Stack Profiles, Revision 6, Document No. 08-0006-06
[R4] ZigBee Cluster Library Specification, Revision 5, Document No. 07-5123-05
[R5] ZigBee Home Automation Public Application Profile 1.2, Revision 29, Document No. 05-3520-29
[R6] ZigBee Smart Energy Standard 1.1b, Revision 18, Document No. 07-5356-18

Device Anatomy

The application controlling the ZigBee/USB Adapter U1 has complete control over the ZigBee anatomy of the device, except for endpoint #0, which hosts the ZigBee Device Object, and certain reserved endpoints:

Endpoint #	Profile	Application
0 (0x00)	0x0000: ZigBee Device Profile	ZigBee Device Object (ZDO) – standard management features
1 (0x01) …239 (0xEF)	0x0104: ZigBee Common Profile (typically, set by host application)	Defined by host application
240 (0xF0)	n/a	Reserved
241 (0xF1)	n/a	Reserved
242 (0xF2)	0xA1E0: ZigBee Green Power Profile	ZigBee Green Power Proxy or Combined Proxy and Sink
243 (0xF3) … 254 (0xFE)	n/a	Reserved
255 (0xFF)	n/a	Broadcast endpoint address (broadcast to all endpoints)

The ubisys ZigBee manufacturer ID is 0x10F2. This manufacturer code is used to identify upgrade images, for accessing manufacturer-specific ZCL attributes and commands etc.

Installation Code

This router has a pre-configured link key, which is derived from the installation code printed on the back of the router's housing in text format and as a two-dimensional barcode (QR code). The format specified in [R6], section 5.4.8.1.1 is used with a full 128-bit installation code + 16-bit CRC. The QR code contains additional information as illustrated in the following example:

```
ubisys2/R0/001FEE00000000FF/0F7C1CD805F91649EBA84580AA1CB432F51A/21
```

Here, "ubisys2" is the data format identifier, R0 is the model string (this is just an example, it would be "U1" for this product), followed by the EUI-64, the installation code, and a checksum that covers the entire information (including model, EUI-64 and installation code), all separated by a single dash character ('/'). The check sum is an unsigned 8-bit integer, which is calculated by performing a byte-wise exclusive-or (XOR, "⊕") of the ASCII characters of the model string, the binary representation of the EUI-64 (in big endian format), and the binary representation of the install code.

For the example above, this calculation yields:

```
52 ('R') ⊕ 30 ('0') ⊕
00 ⊕ 1F ⊕ EE ⊕ 00 ⊕ 00 ⊕ 00 ⊕ 00 ⊕ FF ⊕
0F ⊕ 7C ⊕ 1C ⊕ D8 ⊕ 05 ⊕ F9 ⊕ 16 ⊕ 49 ⊕ EB ⊕ A8 ⊕ 45 ⊕ 80 ⊕ AA ⊕ 1C ⊕ B4 ⊕ 32 ⊕ F5 ⊕ 1A = 21
```

Notice: The data format “ubisys1” is identical to “ubisys2” except for the check sum, which is present, but invalid. If you encounter “ubisys1” labels, then do not verify the trailing check sum field.

9.1. Application Endpoint #0 – ZigBee Device Object

Please refer to the ZigBee Specification [R2] for details on the ZigBee Device Object (ZDO) and the protocol used for over-the-air communication, called the ZigBee Device Profile (ZDP). Notice that the ZDP is fundamentally different from typical application endpoints, which build on the ZigBee foundation framework and the ZigBee Cluster Library (ZCL).

The ubisys ZigBee/USB Adapter U1 supports the following ZDO services:

Primitive	Description
nwk_addr_req/ nwk_addr_rsp	**Network address request/response** Translates a 64-bit IEEE address into a 16-bit network short address. Use only when really required, because this message employs a network-wide broadcast (flooding)
ieee_addr_req/ ieee_addr_rsp	**IEEE address request/response** Translates a 16-bit network short address into a 64-bit IEEE address.
node_desc_req/ node_desc_rsp	**Node descriptor request/response** Returns information such as the manufacturer ID, power supply, etc.
power_desc_req/ power_desc_rsp[10]	**Power descriptor request/response** Returns information such as the power source and mode.
active_ep_req/ active_ep_rsp	**Active endpoints request/response** Returns a set of available application endpoints on the device.
simple_desc_req/ simple_desc_rsp	**Simple descriptor request/response** Returns a descriptor for a certain application endpoint with a list of available services (clusters).
match_desc_req/ match_desc_rsp	**Match descriptor request/response** Searches for a certain cluster or set of clusters and returns the matching endpoints, if any.
device_annce	**Device announcement** Advertises the presence of a new device in the network.
parent_annce/ parent_annce_rsp[11]	**Parent announcement/response** This is part of the ZigBee 2015 end-device child management feature.
system_server_discovery_req/ system_server_discovery_rsp[12]	**System server discovery request/response** Provides the ability to discover system servers, in particular the network manager.
bind_req/ bind_rsp	**Bind request/response** Creates an application binding
unbind_req/ unbind_rsp	**Unbind request/response** Removes an application binding
mgmt_nwk_disc_req/ mgmt_nwk_disc_rsp[13]	**Management: Network discovery request/response** Instructs the device to perform a network discovery and report the results back.
mgmt_lqi_req/ mgmt_lqi_rsp	**Management: Neighbor table request/response** Returns information about neighboring devices, including the link quality, device type etc.
mgmt_rtg_req/ mgmt_rtg_rsp	**Management: Routing table request/response** Returns information about routes established on the device.
mgmt_bind_req/ mgmt_bind_rsp	**Management: Binding table request/response** Returns information about application bindings on the device.
mgmt_leave_req/ mgmt_leave_rsp	**Management: Leave request/response** Makes the device leave the network or removes one of its end-device children.
mgmt_permit_joining_req/ mgmt_permit_joining_rsp	**Management: Permit joining request/response** Opens the network for new devices to join.
mgmt_nwk_update_req/ mgmt_nwk_update_notify[14]	**Management: Network update request/response/notification** Performs energy scans, instigates a channel change or assigns the network manager.

[10] Available in ZigBee stack version 1.60 and above. Legacy ZCP requirement – do not use in applications
[11] Available in ZigBee stack version 1.56 and above.
[12] Available in ZigBee stack version 1.50 and above.
[13] Available in ZigBee stack version 1.61 and above.
[14] Available in ZigBee stack version 1.61 and above.

9.2. Application Endpoint #242 – ZigBee Green Power

This endpoint provides the ZigBee Green Power feature according to the 2015 edition of the specification, i.e. including support for Green Power Devices with IEEE EUI-64 and bidirectional commissioning. The ZigBee traffic between Proxies and Sinks utilizes standard ZigBee foundation paradigms and the ZigBee Cluster Library [R4]. You may use the standard ZCL frames to enumerate, read and write attributes, invoke commands, etc.

The application endpoint exposes the following clusters:

Cluster	Direction	Description
0x0021	Outbound (Client)	**ZigBee Green Power Proxy** Allows sinks on the network to configure this device as a "Proxy", i.e. access point for ZigBee Green Power Devices into the ZigBee mesh network.

9.2.1. Green Power Cluster (Client)

The client-side of the Green Power cluster provides the ZigBee Green Power Proxy functionality, i.e. makes the device act as an "access point" for Green Power Devices (GPDs). This implementation supports unidirectional and bidirectional[15] GPDs.

Attributes supported:

Attribute	Type	Description
0x0010	unsigned8, read-only	**gppMaxProxyTableEntries** The number of proxy table entries supported by this device
0x0011	extended raw binary, read-only, persistent	**ProxyTable** Entries in the proxy table create a link between Green Power Devices and Green Power Sinks
0x0016	bitmap24, read-only	**gppFunctionality** Indicates Green Power features and building blocks supported by this device
0x0017	bitmap24, read-only	**gppActiveFunctionality** Allows to disable certain Green Power features on this device
0x0020	bitmap8, persistent	**gpSharedSecurityKeyType** Determines the security key type to use for devices with bidirectional commissioning capabilities, i.e. out-of-the-box individual key, shared GP key, etc.
0x0021	key128, persistent	**gpSharedSecurityKey** The 128-bit AES-CCM* key that is being used to secure Green Power data frames
0x0022	key128, persistent	**gpLinkKey** The 128-bit AES-CCM* key that is being used to deliver keying material to Green Power devices

Cluster commands supported:

Command	Description
0x01	**GP Pairing** Creates, updates or removes proxy table entries
0x02	**GP Proxy Commissioning Mode** Makes the proxy enter commissioning mode for a particular sink, or leave commissioning mode

[15] Bidirectional communication is currently limited to the commissioning stage

0x06	**GP Response** Tunnels GP data frames from a sink to a bidirectional Green Power Device
0x0B	**GP Proxy Table Request** Allows to query the proxy table for a certain Green Power Device or read out the table in chunks

Cluster commands transmitted:

Command	Description
0x00	**GP Notification** Tunnels GP frames from a Green Power Device to one or more sinks or groups of sinks
0x04	**GP Commissioning Notification** Tunnels GP frames from a Green Power Device to a sink in commissioning mode
0x0B	**GP Proxy Table Response** Conveys a set of proxy table entries to a sink or management application

9.2.2. Green Power Cluster (Server)

This ZigBee/USB Adapter is capable of promoting its Green Power Endpoint to a Green Power Combined application. This requires a sink implementation in the host application, which works in tandem with the proxy application running on U1. For details, please contact support@ubisys.de.

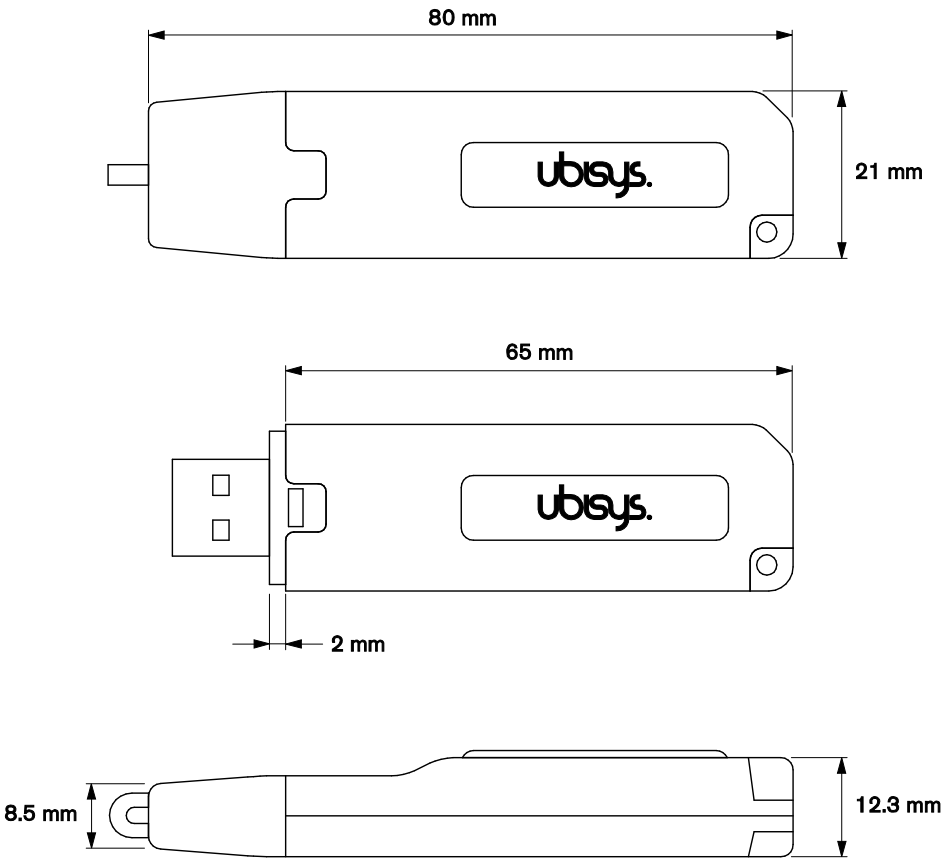

Figure 2: ZigBee/USB Adapter U1 with on-board PCB antenna

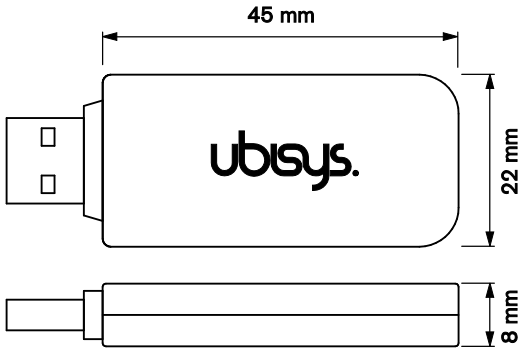

Figure 3: ZigBee/USB Adapter U1-Q with on-board PCB antenna

11. ORDERING INFORMATION

The following tables list the product variants available. Use the specified order code for your orders. Please contact ubisys support if you require any customization.

Case	Firmware variant	Order Code
Black	ZigBee/USB Adapter U1	**9072**
Black	ZigBee/USB Adapter U1-Q	**9096**

12. GENERAL TERMS & CONDITIONS OF BUSINESS

When placing your order you agree to be bound by our General Terms & Conditions of Business, “Allgemeine Geschäftsbedingungen”, which are available for download here: http://www.ubisys.de/en/smarthome/terms.html

13. DECLARATION OF CONFORMITY

CE

We – ubisys technologies GmbH, Am Wehrhahn 45, 40211 Düsseldorf, Germany – declare under our sole responsibility that the ubisys ZigBee/Adapter U1 with order codes as detailed in section 11 under the trade name “ubisys” to which this declaration relates are in conformity with the following directives and standards:

Directive/Standard	Description/Scope
2014/53/EU	Radio Equipment Directive (RED)
2004/108/EC	Electromagnetic Compatibility Directive (EMC)
2006/95/EC	Low Voltage Directive (LVD)
2002/96/EC	Waste Electrical and Electronic Equipment Directive (WEEE)
2002/95/EC	Restriction of Hazardous Substances Directive (RoHS)
EN 300 328	ERM; Wideband transmission systems; 2.4 GHz ISM band
EN 300 440	ERM; Radio equipment to be used in the 1 GHz to 40 GHz frequency range
EN 301 489	EMC
IEEE 802.15.4	IEEE Standard 802 – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)
ZigBee 3.0	ZigBee 2015 with Green Power

Düsseldorf, Germany	April 28, 2016
Place of issue	**Date of issue**
Dr.-Ing. Arasch Honarbacht	Managing Director, Head of Research & Development
Full name of Authorized Signatory	**Title of Authorized Signatory**
	
Signature	**Seal**

14. REVISION HISTORY

Revision	Date	Remarks
1.0	11/14/2014	Initial Public Version
1.1	11/11/2015	Updated the ZDO description to include enhancements and additions made for ZigBee 2015 platform compliance
1.2	09/13/2016	Updated to ZigBee 3.0 Certified Product in application firmware 1.68
1.3	01/03/2017	Added USB protocol description placeholder and 7Bfx™ API documentation
1.4	01/25/2017	Added details about ZigBee/USB Adapter U1-Q, including architecture (SAM4S+GP712+RFX2411), dimensions, and order code. Added system architecture diagram

15. CONTACT

UBISYS TECHNOLOGIES GMBH
AM WEHRHAHN 45
40211 DÜSSELDORF
GERMANY

T: +49 (211) 54 21 55 - 00
F: +49 (211) 54 21 55 - 99

www.ubisys.de
info@ubisys.de