CyBridge user guide

About this document

Scope and purpose

CyBridge is a cross-platform library written in C++. It simplifies the usage of KitProg3, MiniProg4, and UART devices by providing high-level APIs.

The CyBridge can:

- Perform I^2C, SPI, and UART communication
- Control and monitor power
- Provide callbacks about device connect and disconnect.

Intended audience

This document helps you understand how to use the CyBridge interface.

Document conventions

Convention	Explanation
Bold	Emphasizes heading levels, column headings, menus and sub-menus
Italics	Denotes file names and paths.
`Courier New`	Denotes APIs, functions, interrupt handlers, events, data types, error handlers, file/folder names, directories, command line inputs, code snippets
File > New	Indicates that a cascading sub-menu opens when you select a menu item

Table of contents

Table of contents

1 Introduction

1.1 Supported OS

The CyBridge supports the following operating systems:

- Windows 7 SP1 64-bit
- Windows 10 64-bit and newer
- Ubuntu 18.04 LTS Linux 64-bit and newer
- macOS 10, Big Sur 64-bit and newer

1.2 Supported hardware (probes)

The CyBridge library supports these hardware programming devices (probes):

- I^2C, SPI, Connect/Disconnect event:
  - MiniProg4 stand-alone programmer
  - KitProg3 onboard programmer
- Serial communication, connect/disconnect event:
  - All serial devices
- Connect/disconnect event:
  - Kits with AIROC™ Wi-Fi and/or Bluetooth® devices
  - Segger J-Link

1.3 Package contents

The CyBridge package contains:

- CyBridge library – The library described in this guide.
- Example – The QT-based example that shows the functionality of CyBridge. To speed up the usage of the library, review this example before creating your own code.
- *Cybridge.h* File – The header file with a description of all exported functions and their parameters. It should be included in all projects that use the CyBridge library.
- Auxiliary Libraries –Additional libraries used by the CyBridge library.

2 API Description

This section includes all the API commands, their descriptions, and usage examples. The following table lists the available CyBridge API functions:

API	Description
createCyBridge	The factory that creates a CyBridge instance.
initialize	Performs initialization of the CyBridge library. Without the initialization, the library is not operable.
exit	Releases all resources occupied by the CyBridge. It calls "delete this" to release itself.
getVersion	Returns a version of the library.
getErrorDescription	Returns a description of the passed error code.
setConnectCallback	Sets the callback function to be called on each device connect and disconnect.
setLogCallback	Sets the callback function for CyBridge logger's messages. Helps to understand the state of the CyBridge and connected device.
getDeviceList	Returns a list of connected devices.
releaseDeviceList	Releases a list of connected devices allocated by the getDeviceList API.
openDevice	Opens a specific device for operation.
closeDevice	Closes the device opened by the openDevice API. The API call is not mandatory. Added for consistency only.
getDeviceProps	Returns the properties of the opened device.
releaseDeviceProps	Releases the device properties allocated by the getDeviceProps API.
getCyBridgeParam	Returns the parameter value of CyBridge library.
setCyBridgeParam	Allows to set a parameter specific for the whole cybridge library
getPowerMonitorParam	Returns the parameter value of the power monitor associated with the opened device.
getPowerControlParam	Returns the parameter value of the power control associated with the opened device.
getI2cParam	Returns the parameter value of the I^2C interface associated with the opened device.
setI2cParam	Sets the parameter value of the I^2C interface associated with the opened device.
getSpiParam	Returns the parameter value of the SPI interface associated with the opened device.
setSpiParam	Sets the parameter value of the SPI interface associated with the opened device.
getUartParam	Returns the parameter value of the UART interface associated with the opened device.
setUartParam	Sets the parameter value of the UART interface associated with the opened device.
getSwdParam	Returns the parameter value of the SWD interface associated with the opened device.
getJtagParam	Returns the parameter value of the JTAG interface associated with the opened device.
getVoltage	Returns the voltage measured by the opened device.
setVoltage	Sets the voltage by the power-control interface associated with the opened device.
powerOn	Enables/disables the output power.

API Description

API	Description
modeSwitch	Allows transition among modes of the KitProg3 device.
upgradeFw	Upgrades the probe FW.
i2cDataTransfer	Sends or receives data over the I²C protocol.
spiDataTransfer	Sends and receives data over the SPI protocol.
uartWrite	Sends data over the UART protocol.
uartRead	Reads data from the UART protocol until the provided buffer is filled or operation timed out.
uartReadSinglePacket	Reads the single packet data from the UART protocol.
uartReadAsync	Starts/stops the Async read of data from the UART protocol.
usbWrite	Sends data over the USB protocol.
usbWriteRead	Sends raw data over USB protocol and receives response.
i2cStartMultipleTransaction	Starts the multiple transaction mode.
i2cStopMultipleTransaction	Stops the previously started multiple I²C transaction.
i2cAppendCommand	Appends a short I²C command to internal buffer of the current multiple transaction.
i2cExecuteCommands	Initiates the execute commands action.
i2cGetCmdResponse	Returns the response of the I²C command by the given index.
cancelTransfer	Cancels the I²C /SPI data transfer operation
getGPIOParam	Returns the parameter value of the GPIO interface associated with the opened device.
gpioSetMode	Sets drive mode of the Kitprog3 GPIO pin
gpioSetState	Sets state of the KitProg3 GPIO pin
gpioRead	Reads the current state of the KitProg3 GPIO pin
gpioStateChanged	Reads whether the state of the KitProg3 GPIO pin has changed
resetDevice	Issues the KitProg3 device reset request
resetTarget	Issues the target MCU reset request
setUartFlowControl	Sets the UART flow control mode for the requested KitProg3 UART
getUartFlowControl	Retrieves the UART flow control mode set for requested KitProg3 UART
initRtt	Initializes the RTT interface
clearRtt	Destroys the RTT interface
rttConnectSocketToHost	Attempts to make a connection to the host on a given port
rttDisconnect	Closes a connection made to the host on a given port
rttRead	Reads the data available on the RTT interface
rttWrite	Sends a raw data over RTT interface
rttIsActive	Checks if connection between socket and host was established
rttSetReadyReadEnable	Switches on/off RTT Read Data Mode
rttClearData	Clears all internal RTT data buffers

2.1 createCyBridge

The factory that creates a CyBridge instance. It is the first CyBridge API to be called.

Signature:

```
CyBridge* createCyBridge();
```

Example:

```
#include "cybridge.h"

CyBridge* bridge_p = createCyBridge();
if (nullptr == bridge_p) {
  std::cout << "Error: Cannot create CyBridge !" << std::endl;
}
```

2.2 initialize

Performs initialization of the CyBridge library. Without the initialization, the library is not operable.

Note: *On macOS, initialization with the `Bridge_with_UART` parameter, due to specifics of the OS, results in detection of two CyBridge devices: the one with I²C/SPI + UART interfaces, and the second only with UART interface available.*

Signature:

```
int32_t initialize(
    uint32_t devices = 0, CyBridgeProps* props = nullptr)
```

API Description

Arguments:

Devices	The mask of devices to be reported. A few devices can be OR-ed. All masks are listed in the `CyBridgeDevices` enum. By default, all supported devices are reported.
	`/* Defines device types to be reported by CyBridge */`
	`enum CyBridgeDevices : uint32_t {`
	`  /* All devices */`
	`  All = 0x0000,`
	`  /* CMSIS-DAP devices */`
	`  CMSIS_DAP = 0x0001,`
	`  /* All Bridge devices */`
	`  Bridge = 0x0002,`
	`  /* KitProg2 Bridge devices  */`
	`  Bridge_KP2 = 0x0004,`
	`  /* KitProg3 and MiniProg4 Bridge devices */`
	`  Bridge_KP3_MP4 = 0x0008,`
	`  /* UART devices */`
	`  UART = 0x0010,`
	`  /* JLink devices */`
	`  JLink = 0x0020,`
	`  /* WICED WiFi devices */`
	`  WICED_WiFi = 0x0040,`
	`  /* WICED BlueTooth devices */`
	`  WICED_BT = 0x0080,`
	`  /* KitProg bridge with UART devices. Automatically enables`
	`   * Bridge_KP3_MP4 | UART if not enabled */`
	`  Bridge_with_UART = 0x1000,`
	`};`

Props	The pointer to the `CyBridgeProps` structure. E.g. it allows to specify the HID device access mode.
	`/* Defines the properties of the CyBridge library */`
	`struct CyBridgeProps {`
	`  /* If true - changes default HID device access mode from (Exclusive +`
	`Shared) to only Exclusive on Windows 8/10. This property has no effect on`
	`Linux or macOS */`
	`  bool ExclusiveHidAccessMode;`
	`};`

Return:

error code.

Example:

```
CyBridge* bridge_p = createCyBridge();

// Initialize CyBridge to support Bridge and UART devices
CyBridgeProps props {
    false, // ExclusiveHidAccessMode
  };
int ret = bridge_p->initialize(CyBridgeDevices::Bridge | CyBridgeDevices::UART, &props);

if (ret < 0) {
  std::cout << "Cannot initialize CyBridge. Err = " << bridge_p-> getErrorDescription(ret)
<< std::endl;
}
```

2.3 exit

Releases all resources occupied by the CyBridge. It calls "delete this" to release itself.

Signature:

```
void exit()
```

Example:

```
CyBridge* bridge_p = createCyBridge();
...
bridge_p->exit();
```

2.4 getVersion

Returns a version of the CyBridge.

Signature:

```
int32_t getVersion(
    uint32_t& major,
    uint32_t& minor,
    uint32_t& patch,
    uint32_t& build)
```

Arguments:

major	The returned CyBridge major version.
minor	The returned CyBridge minor version.
patch	The returned CyBridge patch version.
build	The returned CyBridge build number.

Return:

error code.

Example:

```
uint32_t major = 0;
uint32_t minor = 0;
uint32_t patch = 0;
uint32_t build = 0;

int ret = bridge_p->getVersion(major, minor, patch, build);
if (ret < 0) {
  std::cout << "Cannot get the version of CyBridge. Err = " << bridge_p-
>getErrorDescription(ret) << std::endl;
}
std::cout << "CyBridge version: " << major << "." << minor << "." << patch << "." << build
<< std::endl;
```

2.5 getErrorDescription

Returns a description of the passed error code.

Signature:

```
const char* getErrorDescription(
    int32_t error)
```

Arguments:

error	The error code from the CyBridgeErrors enum whose description should be returned.

Return:

A description of the passed error code.

Example:

```
uint32_t major = 0;
uint32_t minor = 0;
uint32_t patch = 0;
uint32_t build = 0;

CyBridge* bridge_p = createCyBridge();

int ret = bridge_p->getVersion(major, minor, patch, build);
if (ret < 0) {
  std::cout << "Cannot get the version of CyBridge. Err = " << bridge_p-
>getErrorDescription(ret) << std::endl;
}
```

Here `bridge_p->getErrorDescription` returns `"Initialize API was not called"` as there is no "initialize API" in this example.

2.6 setConnectCallback

Sets the callback function for the Connect Event. The Connect Event is raised on connect and disconnect of supported devices. The supported devices are defined in the *initialize* API.

Signature:

```
void setConnectCallback(
    CyBridgeConnectCallback func,
    void* client_p)
```

Arguments:

`func`	The pointer to the function to be called if a device is connected/disconnected. The function is defined as: `typedef void(__cdecl* CyBridgeConnectCallback)(CyBridgeDevice* device_p, bool is_connected, void* client_p);` Here: • `device_p` – The pointer to the CyBridgeDevice structure with a description of the connected device. • `is_connected` – True if the device is connected, False – if disconnected • `client_p` – The user-supplied data to the `setConnectCallback` API.
`client_p`	The client-supplied data passed to the callback function when called.

Example:

```
// A user's registered callback for device Connect/Disconnect events
void connectEvent(CyBridgeDevice* dev_p, bool is_connected, void* /*client_p*/) {
  if (is_connected) {
    std::cout << "Connected ";
  } else {
    std::cout << "Disconnected ";
  }

  std::cout << "device: " << dev_p->Name << std::endl;
}

// User's code
CyBridge* bridge_p = createCyBridge();
bridge_p->setConnectCallback(&connectEvent, nullptr);
bridge_p->initialize();
```

2.7 setLogCallback

Sets the callback function for the CyBridge Logger. It helps understand the state of the CyBridge and connected device.

Signature:

```
void setLogCallback(
    CyBridgeLogCallback func,
    void* client_p,
    CyBridgeLogLevel level = CyBridgeLogLevel::Info)
```

Arguments:

`func`	The pointer to the function to be called if a log message arrives. The function is defined as: `typedef void(__cdecl* CyBridgeLogCallback)(const char* msg_p, CyBridgeLogLevel level, void* client_p);` Here: • `msg_p` – The received log message. • `Level` – The log level of the received message. • `client_p` – The user-supplied data to the setLogCallback API.
`client_p`	The client-supplied data passed to the callback function when called.

`level`	The max level of log messages to be displayed. The default log level is `Info`.
	The supported values are listed in the `CyBridgeLogLevel` enum:
	`/* Defines the levels of the logger */`
	`enum CyBridgeLogLevel {`
	`  Debug = 1,`
	`  Info = 2,`
	`  Warning = 5,`
	`  Error = 6,`
	`};`

Example:

```
// A user's registered callback for displaying CyBridge logs
void logEvent(const char* msg_p, CyBridgeLogLevel /*level*/, void* /*client_p*/) {
  std::cout << "Log: " << msg_p;
}

// User's code
CyBridge* bridge_p = createCyBridge();
bridge_p->setLogCallback(&logEvent, nullptr, CyBridgeLogLevel::Info);
bridge_p->initialize();
```

2.8 getDeviceList

Returns a list of connected devices. The returned list must be released by the *releaseDeviceList* API.

The ComType field of the returned CyBridgeDevice structure can be used to identify UART devices. In this case it must be equal to "serial".

Signature:

```
int32_t getDeviceList(
    CyBridgeDevice*** list_p)
```

Arguments:

list_p	The pointer to return a list of CyBridgeDevice structures that is defined as:
	`/* Defines the connected device */`
	`struct CyBridgeDevice {`
	`/* The device name */`
	`const char* Name;`
	`/* The short device name */`
	`const char* ShortName;`
	`/* The product ID */`
	`uint16_t Pid;`
	`/* The vendor ID */`
	`uint16_t Vid;`
	`/* The device serial number */`
	`const char* SerialNumber;`
	`/* The product name */`
	`const char* Product;`
	`/* The manufacturer name */`
	`const char* Manufacturer;`
	`/* The USB location */`
	`const char* Location;`
	`/* The name of the probe to which the current device belongs */`
	`const char* ProbeName;`
	`/* The communication type: hid, bulk, serial */`
	`const char* ComType;`
	`/* The operation speed in bps - used only by Wiced Bluetooth devices */`
	`uint32_t OperationSpeed;`
	`};`
	
	If nullptr is passed, then only the number of the connected devices is returned.

Return:

If >= 0 – the number of connected devices, otherwise – error code.

Example:

```
CyBridge* bridge_p = createCyBridge();
bridge_p->initialize();
CyBridgeDevice** dev_list = nullptr;
int count = bridge_p->getDeviceList(&dev_list);
std::cout << "The number of connected devices = " << count << std::endl;
for (int i = 0; i < count; i++) {
  std::cout << "\nN = " << i + 1 << " Name = " << dev_list[i]->Name << ", vid = 0x" <<
std::hex << dev_list[i]->Vid << ", pid = 0x" << dev_list[i]->Pid << std::dec << ", SN = "
<< dev_list[i]->SerialNumber << ", Manufacturer = " << dev_list[i]->Manufacturer << ",
Product = " << dev_list[i]->Product << ", Location = " << dev_list[i]->Location << ",
Supported kits = " << dev_list[i]->SupportedKits << ", Probe name = " << dev_list[i]-
>ProbeName << ", Communication = " << dev_list[i]->ComType << ", Operation speed = " <<
dev_list[i]->OperationSpeed << std::endl;
}

bridge_p->releaseDeviceList(dev_list);
```

2.9 releaseDeviceList

Releases a list of connected devices returned by the getDeviceList API.

Signature:

```
int32_t releaseDeviceList(
    CyBridgeDevice** list_p)
```

Arguments:

list_p	The pointer to the list of CyBridgeDevice structures returned by the getDeviceList API.

Return:

error code.

Example:

See an example of the getDeviceList API.

2.10 openDevice

Opens a specific device for operation. The list of connected devices can be obtained via the getDeviceList API.

Signature:

```
int32_t openDevice(
    const char* device_name_p)
```

Arguments:

device_name_p	The pointer to the name of the device to be opened.

Return:

error code.

Example:

```
CyBridgeDevice** dev_list = nullptr;
int count = bridge_p->getDeviceList(&dev_list);
if (count > 0)
  int ret = bridge_p->openDevice(dev_list[0]->Name);

bridge_p->releaseDeviceList(dev_list);
```

2.11 closeDevice

Closes the device opened by the openDevice API. The API call is not mandatory. Added for the consistency only.

Signature:

```
void closeDevice()
```

Example:

```
CyBridgeDevice** dev_list = nullptr;
int count = bridge_p->getDeviceList(&dev_list);
if (count > 0) {
  bridge_p->openDevice(dev_list[0]->Name);
  ...
  bridge_p->closeDevice();
}
bridge_p->releaseDeviceList(dev_list);
```

2.12 getDeviceProps

Gets the properties of the device opened by the openDevice API. The returned properties must be released by the releaseDeviceProps API.

The API can be used to filter devices that support a specific interface, for example, I²C, SPI, etc. It is not recommended to use the API to check if a device supports the UART protocol, because the device must be opened and opening of UART device on macOS can take a while. Instead use the ComType field of CyBridgeDevice structures – if ComType is equal to "serial" then it is a UART device. The CyBridgeDevice structure can be obtained via getDeviceList API or a Connect Callback. Information from KitProg3 Unique ID (UID) Record may be obtained for KitProg3 Device, only if UID is stored on the device. Features from KitProg3 UID Record are only available with KitProg3 firmware major version 2 or higher.

Signature:

```
int32_t getDeviceProps(
    CyBridgeDeviceProps** props_p)
```

API Description

Arguments:

props_p	The pointer to the CyBridgeDeviceProps structure defined as:
	`/* Defines the properties of the connected device */`
	`struct CyBridgeDeviceProps {`
	`  /* The device USB characteristics */`
	`  CyBridgeDevice Device;`
	`  /* The number of supported interfaces */`
	`  uint8_t NumIfs;`
	`  /* The comma-separated list of supported interfaces */`
	`  const char* ListIfs;`
	`  /* The major firmware version */`
	`  uint32_t FwVerMaj;`
	`  /* The minor firmware version */`
	`  uint32_t FwVerMin;`
	`  /* The FW build number */`
	`  uint32_t FwBuildNumber;`
	`  /* The hardware ID */`
	`  uint32_t HwId;`
	`  /* The major protocol version */`
	`  uint32_t ProtVerMaj;`
	`  /* The minor protocol version */`
	`  uint32_t ProtVerMin;`
	`  /* If this field contains 'true' the KitProg3 UID is available and`
	`  the next field Kp3UidInfo contains valid data describing`
	`  capabilities of KitProg3 device and the board itself */`
	`  bool UidIsPresent;`
	`  /* Information about device from KitProg3 UID */`
	`  CyBridgeKitProg3UidDeviceInfo Kp3UidInfo;`
	`};`

Return:

error code.

Example:

```
CyBridgeDevice** dev_list = nullptr;
int count = bridge_p->getDeviceList(&dev_list);
if (count > 0) {
  bridge_p->openDevice(dev_list[0]->Name);
  CyBridgeDeviceProps* props;
  bridge_p->getDeviceProps(&props);
  ...
  bridge_p->releaseDeviceProps(props);
}

bridge_p->releaseDeviceList(dev_list);
```

2.13 releaseDeviceProps

Releases the device properties returned by the getDeviceProps API.

Signature:

```
int32_t releaseDeviceProps(
    CyBridgeDeviceProps* props_p)
```

Arguments:

props_p	The pointer to the device properties to be released.

Return:

error code.

Example:

See the getDeviceProps API example.

2.14 getCyBridgeParam

Gets the parameter value of the CyBridge library. The parameter value is returned as a string.

Signature:

```
int32_t getCyBridgeParam (
    CyBridgeGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len)
```

Arguments:

param	The specific parameter of the CyBridgeGetParams enum to be returned.
out_buff_p	The pointer to the buffer to store the parameter value in.
out_buff_len	The size of the buffer pointed by the out_buff_p argument

Return:

If >= 0 number of bytes in out_buff_p filled by the return value, otherwise – error code.

2.15 setCyBridgeParam

This API allows to set a specific parameter applicable to the CyBridge library itself. E.g. it allows to disable log messages of a particular level.

Signature:

```
int32_t setCyBridgeParam (
    CyBridgeSetParams param,
    uint32_t value)
```

Arguments:

param	The specific parameter of the `CyBridgeSetParams` enum to be returned.
	The enum is defined as:
	`/* Defines the set parameters of the CyBridge library */`
	`enum CyBridgeSetParams: uint8_t {`
	`DisableAllLogs,`
	`DisableWarningLogs,`
	`DisableErrorLogs,`
	`DisableInfoLogs,`
	`};`

value	The value of the parameter to be set

Return:

error code.

Example:

```
bridge_p-> setCyBridgeParam ( CyBridgeSetParams::DisableAllLogs, true);
```

2.16 getPowerMonitorParam

Gets the parameter value of a power monitor associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getPowerMonitorParam(
    CyBridgePowerMonitorGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgePowerMonitorGetParams` enum to be returned.
	The enum is defined as:
	`/* Defines the get parameters of the power-monitor interface */`
	`enum CyBridgePowerMonitorGetParams : uint8_t {`
	`/* Defines if the power control supports the voltage measurement. Contains the "true" or "false" values */`
	`CanMeasureVoltage = 0`
	`};`

out_buff_p	The pointer to the buffer to store the parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument
interface_num	The number of the power monitor interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by the return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p->getPowerMonitorParam( CyBridgePowerMonitorGetParams::CanMeasureVoltage, buff,
BufferSize);
if (std::string(buff) == "true")
  std::cout << "Can Measure Voltage << std::endl;
```

2.17 getPowerControlParam

Gets the parameter value of a power control associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getPowerControlParam(
    CyBridgePowerControlGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgePowerControlGetParams` enum to be returned.
	The enum is defined as:
	`/* Defines the get parameters of the power control interface */`
	`enum CyBridgePowerControlGetParams : uint8_t {`
	`/* Defines if the power on/off is supported. Contains the "true" or "false" values */`
	`IsOnOffSupported = 0,`
	`/* Defines if the voltage control is supported. Contains the "true" or "false" values */`
	`IsVoltCtrlSupported,`
	`/* Defines if the power control is continuous. Contains the "true" or "false" values */`
	`IsContinuous,`
	`/* The min supported voltage in mV */`
	`MinVoltMv,`
	`/* The max supported voltage in mV*/`
	`MaxVoltMv,`
	`/* The current set voltage in mV*/`
	`LastSetVoltMv,`
	`/* The comma-separated list of supported voltages for discreet power controls.`
	`The list is empty for the continuous power controls*/`
	`DiscreteVoltListMv,`
	`/* "true" if this interface powers the board,`
	`"false" if an external supply powers the board*/`
	`IsPoweredByThisInterface`
	`};`

out_buff_p	The pointer to the buffer to store a parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument.
interface_num	The number of a power control interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p-> getPowerControlParam(CyBridgePowerControlGetParams::MinVoltMv, buff,
BufferSize);
std::cout << "Min voltage = " << buff;

bridge_p-> getPowerControlParam(CyBridgePowerControlGetParams::MaxVoltMv, buff,
BufferSize);
std::cout << ", Max voltage = " << buff << std::endl;
```

2.18 getI2cParam

Gets the parameter value of the I²C interface associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getI2cParam(
    CyBridgeI2cGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeI2cGetParams` enum to be returned.
	The enum is defined as:
	`/* Defines the get parameters of the I2C interface */`
	`enum CyBridgeI2cGetParams : uint8_t {`
	`/* The list of supported frequencies in Hz */`
	`I2cFreqListHz = 0,`
	`/* The currently selected frequency*/`
	`CurrentI2cFreqHz = 1,`
	`/* Checks if KP3 firmware supports the I2C multiple transaction mode. Returns "true" if yes*/`
	`IsMultipleTransactionSupported = 2,`
	`/* Returns number of available bytes for the next single command to be appended to the i2c multiple transaction. Note: The number of available bytes is associated only with a single i2c command's payload. You must not assume that several commands may fit into this buffer */`
	`BytesAvailableForNextCmd = 3,`
	`/* Returns current state of the multiple I2C transaction. Possible values are 0 - stopped, 1 - started, 2 - completed, see corresponding CyBridgeI2cMultipleTransactionState enum */`
	`I2cTransactionState = 4,`
	`/* Checks whether current transaction contains commands to be executed */`
	`HasCommandsToExecute = 5,`
	`/* Returns "true" if i2c command can be appended to the multiple transaction buffer */`
	`CanAppendCommand = 6`
	`};`

out_buff_p	The pointer to the buffer to store a parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument.
interface_num	The number of the I²C interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by a return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p->getI2cParam(CyBridgeI2cGetParams::I2cFreqListHz, buff, BufferSize);

std::cout << "Supported I2C frequencies, Hz = " << buff << std::endl;
```

2.19 setI2cParam

Sets the parameter value of the I²C interface associated with the opened device.

Signature:

```
int32_t setI2cParam(
    CyBridgeI2cSetParams param,
    uint32_t value,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeI2cSetParams` enum to be set.
	The enum is defined as:
	`/* Defines the set parameters of the I2C interface */`
	`enum CyBridgeI2cSetParams : uint8_t {`
	`  /* The I2C frequency in Hz */`
	`  I2cFreqHz = 0,`
	`  /* Disables the I2C hardware, waits for the bus to be released, and re-enables the I2C hardware */`
	`  RestartI2cMaster`
	`};`

value	The value of the parameter to be set.
interface_num	The number of the I²C interface whose properties should be set. By default, a zero interface is used.

Return:

error code.

Example:

```
bridge_p->setI2cParam(CyBridgeI2cSetParams::I2cFreqHz, 100000);
```

2.20 getSpiParam

Gets the parameter value of the SPI interface associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getSpiParam(
    CyBridgeSpiGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeSpiGetParams` enum to be returned.
	The enum is defined as:
	`/* Defines the get parameters of the SPI interface */`
	`enum CyBridgeSpiGetParams : uint8_t {`
	`/* The list of the supported predefined frequencies in Hz.`
	`* Use the IsSpiFreqPredefined parameter to check if the device supports such a list of frequencies.`
	`* If not, then the min and max frequencies are returned */`
	`SpiFreqListHz = 0,`
	`/* The currently selected frequency */`
	`CurrentSpiFreqHz = 1,`
	`/* SPI Mode: 0-3 */`
	`SpiModeGet = 2,`
	`/* Bit Order (MSB=0, LSB=1) */`
	`SpiBitOrderGet = 3,`
	`/* Defines if the device supports the pre-defined SPI frequencies.`
	`* If supports, then the SpiFreqListHz parameter contains a list of this frequencies.`
	`* If not, then the min and max frequencies are returned by the SpiFreqListHz parameter*/`
	`IsSpiFreqPredefined = 4`
	`};`

out_buff_p	The pointer to the buffer to store a parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument.
interface_num	The number of the SPI interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by a return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p->getSpiParam(CyBridgeSpiGetParams::CurrentSpiFreqHz, buff, BufferSize);

std::cout << "Current SPI Frequency, Hz = " << buff << std::endl;
```

2.21 setSpiParam

Sets the parameter value of the SPI interface associated with the opened device.

Signature:

```
int32_t setSpiParam(
    CyBridgeSpiSetParams param,
    uint32_t value,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeSpiSetParams` enum to be set.
	The enum is defined as:
	`/* Defines the set parameters of the SPI interface */`
	`enum CyBridgeSpiSetParams : uint8_t {`
	`/* The SPI frequency in Hz */`
	`SpiFreqHz = 0,`
	`/* SPI mode (0-3) */`
	`SpiModeSet = 1,`
	`/* The SPI bit order (MSB first = 0, LSB first =1) */`
	`SpiBitOrderSet = 2,`
	`/* The SPI SS*/`
	`SpiSlaveSelect = 3,`
	`};`

value	The value of the parameter to be set.
interface_num	The number of the SPI interface whose properties should be set. By default, a zero interface is used.

Return:

error code.

Example:

```
bridge_p->setSpiParam(CyBridgeSpiSetParams::SpiFreqHz, 120000);
```

2.22 getUartParam

Gets the parameter value of the UART interface associated with the opened device. The parameter value is returned as a string. For example, it allows to obtain the list of supported baud rates for the KP3 device.

Signature:

```
int32_t getUartParam(
    CyBridgeUartGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeUartGetParams` enum to be returned. The enum is defined as: `enum CyBridgeUartGetParams : uint8_t { /* The list of supported baud rates. If this list is empty, the supported baud rates are not specified for the device and client code may choose any baud rate valid for its platform */ BaudRates = 0, };`
out_buff_p	The pointer to the buffer to store a parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument.
interface_num	The number of the UART interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by a return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p->getUartParam(CyBridgeUartGetParams::BaudRates, buff, BufferSize);

std::cout << " Supported UART baud rates = " << buff << std::endl;
```

2.23 setUartParam

Sets the parameter value of a the UART interface associated with the opened device.

Signature:

```
int32_t setUartParam(
    CyBridgeUartSetParams param,
    uint32_t value)
```

Arguments:

param	The specific parameter of the `CyBridgeUartSetParams` enum to be set.
	The enum is defined as:
	`/* Defines the set parameters of the UART interface */`
	`enum CyBridgeUartSetParams : uint8_t {`
	`  /* Baud Rate */`
	`  BaudRate = 0,`
	`  /* The number of the stop bits */`
	`  StopBits = 1,`
	`  /* The parity type */`
	`  ParityType = 2,`
	`  /* The number of the data bits. The supported values are in the range from 5 to 8 */`
	`  DataBits = 3,`
	`};`

value	The value of the parameter to be set.

Note: *The supported baud rate depends on the HW and OS being used.*

Return:

error code.

Example:

```
bridge_p->setUartParam(CyBridgeUartSetParams::BaudRate, 115200);
```

2.24 getSwdParam

Gets the parameter value of the SWD interface associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getSwdParam(
     CyBridgeSWDJtagGetParams param,
     char* out_buff_p,
     uint32_t out_buff_len,
     uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeSWDJtagGetParams` enum to be returned. The enum is defined as: `enum CyBridgeSWDJtagGetParams : uint8_t { /* The list of the supported frequencies in Hz. If strlen(out_buff_p) == 0, then the interface does not support configurable frequencies */ SWDJtagFreqListHz = 0, /* The supported reset types. Any of "software" or "hardware", or both */ ResetType = 1, };`
out_buff_p	The pointer to the buffer to store a parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument.
interface_num	The number of the SWD interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by a return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p->getSwdParam(CyBridgeSWDJtagGetParams::SWDJtagFreqListHz, buff, BufferSize);

std::cout << " Supported SWD frequencies, Hz= " << buff << std::endl;
```

2.25 getJtagParam

Gets the parameter value of the JTAG interface associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getJtagParam(
    CyBridgeSWDJtagGetParams param,
    char* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

param	The specific parameter of the `CyBridgeSWDJtagGetParams` enum to be returned. The enum is defined as: `enum CyBridgeSWDJtagGetParams : uint8_t { /* The list of the supported frequencies in Hz. If strlen(out_buff_p) == 0, then the interface does not support configurable frequencies */ SWDJtagFreqListHz = 0, /* The supported reset types. Any of "software" or "hardware", or both */ ResetType = 1, };`
out_buff_p	The pointer to the buffer to store a parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument.
interface_num	The number of the JTAG interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by a return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p->getJtagParam(CyBridgeSWDJtagGetParams::SWDJtagFreqListHz, buff, BufferSize);

std::cout << " Supported JTAG frequencies, Hz= " << buff << std::endl;
```

2.26 getVoltage

Returns the voltage measured by the opened device in mV.

Signature:

```
int32_t getVoltage(
    int32_t *measured_voltage_p,
    uint8_t interface_num = 0)
```

Arguments:

measured_voltage_p	The measured voltage.
interface_num	The number of the power meter interface that should measure voltage. By default, a zero interface is used.

Return:

error code.

Example:

```
int measured_voltage;
bridge_p->getVoltage(measured_voltage)

std::cout << " Measured voltage = " << measured_voltage << std::endl;
```

2.27 setVoltage

Sets the voltage in mV by the power control interface associated with the opened device.

Signature:

```
int32_t setVoltage(
    uint32_t mV,
    uint8_t interface_num = 0)
```

Arguments:

mV	The voltage in mV to be set.
interface_num	The number of the power control interface that should set the voltage. By default, a zero interface is used.

Return:

error code.

Example:

```
bridge_p->setVoltage(3300);
```

2.28 getGPIOParam

Gets the parameter value of a GPIO Bridge interface associated with the opened device. The parameter value is returned as a string.

Signature:

```
int32_t getGPIOParam(
CyBridgeGPIOGetParams param,
char *out_buff_p, uint32_t out_buff_len,
       uint8_t interface_num = 0);
```

Arguments:

param	The specific parameter of the `CyBridgeGPIOGetParams` enum to be returned.
	The enum is defined as:
	`/* Defines the get parameters of the GPIO Bridge interface */`
	`enum CyBridgeGPIOGetParams : uint8_t {`
	`  HaveGPIO = 0`
	`};`

out_buff_p	The pointer to the buffer to store the parameter value in.
out_buff_len	The size of the buffer pointed by the `out_buff_p` argument
interface_num	The number of the gpio bridge interface whose properties should be returned. By default, a zero interface is used.

Return:

If >= 0 number of bytes in `out_buff_p` filled by the return value, otherwise – error code.

Example:

```
const size_t BufferSize = 5000;
char buff[BufferSize];

bridge_p-> getGPIOParam( CyBridgeGPIOGetParams::HaveGPIO , buff, BufferSize);
if (std::string(buff) == "true")
  std::cout << "Device has GPIO Bridge << std::endl;
```

2.29 powerOn

Enables/disables the output power.

Signature:

```
int32_t powerOn(
    bool onOff,
    uint8_t interface_num = 0)
```

Arguments:

onOff	Defines if the power should be On or Off
interface_num	The number of the power control interface to toggle the power. By default, a zero interface is used.

Return:

error code.

Example:

```
bridge_p->powerOn(true)
```

2.30 modeSwitch

Allows transition among modes of the opened KitProg3-based devices. It includes USB re-enumeration. Supported modes are KitProg3 Bootloader, KitProg3 CMSIS-DAP BULK, KitProg3 CMSIS-DAP HID, KitProg3 DAPLink, KitProg3 BULK with Dual Uart.

Signature:

```
int32_t modeSwitch(
    CyBridgeKitProgModeSwitch mode)
```

Arguments:

mode	The specific parameter of the `CyBridgeKitProgModeSwitch` enum to be provided.
	The enum is defined as:
	`/* Defines the supported KitProg modes */`
	`enum CyBridgeKitProgModeSwitch : uint8_t {`
	`/* KitProg3 Bootloader mode */`
	`KitProg3Bootloader = 0,`
	`/* KitProg3 CMSIS-DAP BULK mode */`
	`KitProg3Bulk = 1,`
	`/* KitProg3 CMSIS-DAP HID mode */`
	`KitProg3Hid = 2,`
	`/* KitProg3 DAPLink mode */`
	`KitProg3DAPLink = 3,`
	`/* KitProg3 Daul UART mode */`
	`KitProg3DualUart = 4,`
	`};`

Return:

error code.

Example:

```
bridge_p->modeSwitch(CyBridgeKitProgModeSwitch::KitProg3Bulk);
```

2.31 upgradeFw

This API is intended only for internal use. Use the fw-loader tool to upgrade firmware of KitProg2-based/KitProg3-based devices instead of this API.

Upgrades the FW of the opened device. It closes the opened device before an FW upgrade.

Signature:

```
int32_t upgradeFw(const char* fw_file_p)
```

Arguments:

`fw_file_p`	The following are the possible options: • The path to the firmware upgrade file. • The content of the whole programming file, which is passed as a null-terminated string. The string has the following format: <"File in buffer">:<format of the file>:<file content>. For example, for the passed cyacd file, the string is: "File in buffer:cyacd:2E12706900....." *Note:* *Only the cyacd file format is supported for passing via a null-terminated string.*

Return:

error code.

Example:

```
bridge_p-> upgradeFw("c:/firmware/kitprog3.cyacd");
```

2.32 i2cDataTransfer

Sends or receives data over the I²C protocol.

Signature:

```
int32_t i2cDataTransfer(
    uint8_t addr,
    uint8_t mode,
    uint32_t out_len,
    const uint8_t* in_buff_p,
    uint32_t in_buff_len,
    uint8_t* out_buff_p,
    uint32_t out_buff_len,
    uint8_t interface_num = 0)
```

Arguments:

addr	The 7-bit address of the I²C slave device.

mode	The mask bits that define the I²C bus protocol signals. Different masks can be ORed. Listed in the CyBridgeI2CMode enum:
	`/* I2C Modes. Passed as the mode argument to the i2cDataTransfer API. Can be ORed together */`
	`enum CyBridgeI2CMode : uint8_t {`
	`/* The current operation is I2C Write */`
	`Write = 0x00,`
	`/* The current operation is I2C Read */`
	`Read = 0x01,`
	`/* Generate the I2C Start condition */`
	`GenStart = 0x02,`
	`/* Generate the I2C Restart condition */`
	`GenReStart = 0x04,`
	`/* Generate the I2C Stop condition */`
	`GenStop = 0x08`
	`};`

out_len	The number of bytes to be read from a slave device. This parameter is used only for Read operation. The value should not exceed (out_buff_len - 1).
in_buff_p	The data to send to the I²C slave device. This parameter is used only for Write operation. • Its size should be equal to or bigger than write_len. • For Read operation, this array is disregarded and its size can be set to 0.
in_buff_len	The number of bytes to send to the I²C slave device. This parameter is used only for Write operation.
out_buff_p	Data returned as a result of transaction. It has different meanings for Read and Write operations. The only common meaning is the first byte—the acknowledgment result of the address byte (1 - ACK, 0 - NACK). Its size should be equal to or bigger than write_len. • Read operation – Contains (1 + out_len) bytes, where out_len is the bytes actually read from the device. They are meaningful if the address byte is ACKed. • Write operation – Contains (1 + in_buff_len) bytes. The second part of the packet contains the ACK / NACK result for every sent byte. Normally, when the first NACK byte occurs, all others are NACKed as well.
out_buff_len	The size of the out buffer.
interface_num	The number of the I²C interface to be used. By default, a zero interface is used.

Return:

error code.

API Description

Examples:

I²C Write example

```
const uint8_t I2CAddr = 7;
uint32_t num_bytes_to_write = 8;
std::vector<uint8_t> write_buff = {1, 2, 3, 4, 5, 6, 7, 8};
uint32_t response_buff_size = num_bytes_to_write + 1;
std::vector<uint8_t> response_buff(response_buff_size);
bridge_p->i2cDataTransfer(
    I2CAddr,
    CyBridgeI2CMode::Write|CyBridgeI2CMode::GenStart|CyBridgeI2CMode::GenStop,
    0,
    &write_buff[0],
    num_bytes_to_write,
    &response_buff[0],
    response_buff_size);

std::stringstream print_str;
for (size_t index = 0; index < num_bytes_to_write; index++) {
  print_str << std::hex << std::setfill('0') << std::setw(2) << std::uppercase <<
static_cast<int>(write_buff[index]) << (response_buff[index + 1] ? "+" : "-") << " ";
}

std::cout << "Addr " << (response_buff[0] ? "ACK" : "NACK") << ", Write data = " <<
print_str.str() << std::endl;
```

I²C Read example

```
const uint8_t I2CAddr = 7;
uint32_t num_bytes_to_read = 8;
uint32_t response_buff_size = num_bytes_to_read + 1;
std::vector<uint8_t> response_buff(response_buff_size);

bridge_p->i2cDataTransfer(
    I2CAddr,
    CyBridgeI2CMode::Read|CyBridgeI2CMode::GenStart|CyBridgeI2CMode::GenStop,
    num_bytes_to_read,
    nullptr,
    0,
    &response_buff[0],
    response_buff_size);

std::stringstream print_str;
for (size_t index = 0; index < num_bytes_to_read; index++) {
  print_str << std::hex << std::setfill('0') << std::setw(2) << std::uppercase <<
static_cast<int>(response_buff[index + 1]) << " ";
}

std::cout << " Addr " << (response_buff[0] ? "ACK" : "NACK") << ", Read data = " <<
print_str.str() << std::endl;
```

2.33 spiDataTransfer

Sends and receives data over the SPI protocol.

Signature:

```
int32_t spiDataTransfer(
    uint8_t mode,
    const uint8_t* in_buff_p,
    uint8_t* out_buff_p,
    uint32_t buff_len,
    uint8_t interface_num = 0)
```

Arguments:

mode	The mask bits that define the SPI bus protocol signals. Different masks can be ORed. Listed in the CyBridgeSPIMode enum:
	`/* SPI Modes. Passed as the mode argument to the spiDataTransfer API. Can be ORed together */`
	`enum CyBridgeSPIMode : uint8_t {`
	`  /* Generates the High->Low transition on the SS pin before a data transfer */`
	`  AassertSSOnStart = 0x01,`
	`  /* Generates the Low->High transition on the SS pin after a data transfer */`
	`  DeassertSSOnStop = 0x02,`
	`};`

in_buff_p	The data to send to the SPI slave device.
out_buff_p	The data returned as a result of transaction.
buff_len	The number of bytes to transfer. The In and Out buffers should be equal to or bigger than the number.
interface_num	The number of the SPI interface to be used. By default, a zero interface is used.

Return:

error code.

Example:

```
uint8_t num_bytes_to_transfer = 8;
std::vector<uint8_t> write_buff= {1, 2, 3, 4, 5, 6, 7, 8};
std::vector<uint8_t> response_buff(num_bytes_to_transfer);

bridge_p->spiDataTransfer(
    CyBridgeSPIMode::AassertSSOnStart|CyBridgeSPIMode::DeassertSSOnStop,
    &write_buff[0],
    &response_buff[0],
    num_bytes_to_transfer);
```

2.34 uartWrite

Sends data over the UART protocol.

Signature:

```
int32_t uartWrite(
    const uint8_t* buff_p,
    uint32_t buff_len)
```

Arguments:

buff_p	The pointer to the data to send.
buff_len	The number of the bytes to send.

Return:

error code.

Example:

```
std::vector<uint8_t> uart_write_data{0x30, 0x31, 0x32, 0x20};

bridge_p->uartWrite(
    &uart_write_data[0],
    static_cast<uint32_t>(uart_write_data.size()));
```

2.35 uartRead

Reads data from the UART protocol until the provided buffer is filled. It is a blocking function that does not exit until a timeout or receipt of the expected number of bytes

Signature:

```
int32_t uartRead(
    uint8_t* buff_p,
    uint32_t buff_len,
    uint32_t timeout_ms = 1000)
```

Arguments:

buff_p	The pointer to the Read buffer.
buff_len	The number of bytes to read, must be <= of the size of the buff_p buffer.
timeout_ms	The timeout for waiting for Read data. The default value is 1000 ms.

Return:

If >= 0 number of read bytes, otherwise – error code.

Example:

```
std::vector<uint8_t> uart_read_data(10);
bridge_p->uartRead(
    &uart_read_data[0],
    static_cast<uint32_t>(uart_read_data.size()));
```

2.36 uartReadSinglePacket

Reads the single packet data from the UART protocol. It is a blocking function that does not exit until a timeout or receipt of a packet of data. The size of a received data packet for the same UART sender can vary on different operating systems from several bytes to several hundred bytes.

Note: *This function may read less bytes than you requested in `buff_len` parameter.*

Signature:

```
int32_t uartReadSinglePacket(
    uint8_t* buff_p,
    uint32_t buff_len,
    uint32_t timeout_ms = 1000)
```

Arguments:

`buff_p`	The pointer to the Read buffer.
`buff_len`	The size of the `buff_p` buffer.
`timeout_ms`	The timeout for waiting for Read data. The default value is 1000 ms.

Return:

If >= 0 number of read bytes, otherwise – error code.

Example:

```
std::vector<uint8_t> uart_read_data(10);
bridge_p->uartReadSinglePacket(
    &uart_read_data[0],
    static_cast<uint32_t>(uart_read_data.size()));
```

2.37 uartReadAsync

Starts/stops the Async Read of data from the UART protocol.

Signature:

```
int32_t uartReadAsync(
    CyBridgeUartRxCallback func,
    uint8_t* buff_p,
    uint32_t buff_len,
    void* client_p)
```

Arguments:

func	The function to be called when new data arrives. If nullptr is passed, then the Async Read is stopped. The function is defined as: `typedef void(__cdecl* CyBridgeUartRxCallback)(uint8_t const* data_p, uint32_t num_bytes, void* client_p);` Where: • data_p – The pointer to received data. • num_bytes – The number of received bytes in data_p buffer. • client_p – The user-supplied data to the setUartRxCallback API.
buff_p	The buffer for Read data.
buff_len	The size of the buff_p buffer.
client_p	The client-supplied data passed to the callback function when called.

Return:

error code.

Example:

```
void uartRxCallback(uint8_t const* data_p, uint32_t num_bytes, void* /*client_p*/) {
  std::stringstream print_str;
  for (uint32_t index = 0; index < num_bytes; index++) {
    print_str << std::hex << std::setfill('0') << std::setw(2) << std::uppercase <<
static_cast<int>(data_p[index]) << " ";
  }

  std::cout << "Uart Read data = " << print_str.str() << std::endl;
  }

std::vector<uint8_t> uart_read_data(10);

// Start the UART Read Async
bridge_p->uartReadAsync(&uartRxCallback, &uart_read_data[0],
static_cast<uint32_t>(uart_read_data.size()), nullptr);

...
// Stop the UART Read Async
bridge_p->uartReadAsync(nullptr, nullptr, 0, nullptr);
```

2.38 usbWrite

Sends raw data over USB protocol.

Signature:

```
int32_t usbWrite(
const uint8_t* buff_p,
uint32_t buff_len)
```

Arguments:

buff_p	The pointer to the data to send.
buff_len	The number of the bytes to send.

Return:

error code.

Example:

```
char hex_data[] = "0x8301"; // Turn On programming LED

bridge_p->usbWrite(
    hex_data,
    static_cast<uint32_t>( strlen(hex_data));
```

2.39 usbWriteRead

Sends raw data over USB protocol and receives response.

Signature:

```
int32_t usbWriteRead(const char* in_buff_p,
uint32_t in_buff_len, uint8_t* out_buff_p,
uint32_t out_buff_len)
```

Arguments:

in_buff_p	The pointer to the data to send.
in_buff_len	The number of the bytes to send.
out_buff_p	The pointer to response data to be received
out_buff_len	The number of bytes to be received

Return:

error code.

Example:

```
std::string hex_data = "0x90"; // Get Info
    size_t out_data_size = 64u;
    std::vector<uint8_t> out_data(out_data_size);
    ret = bridge_p->usbWriteRead(hex_data.c_str(), static_cast<uint32_t>(hex_data.size()),
&out_data[0], out_data_size);
    if (ret < 0) {
        std::cout << "Send USB data failed" << std::endl;
    }
    std::cout << "Response: ";
    for (auto& i: out_data)
    std::cout  << "0x" << std::hex << std::setfill('0') << std::setw(2) <<  std::uppercase
<< static_cast<int>(i) << ' ';
```

2.40 i2cStartMultipleTransaction

Starts the multiple transaction mode. This call initializes the internal library state and allows to start appending I²C commands.

Signature:

```
int32_t i2cStartMultipleTransaction(uint8_t interface_num = 0)
```

Arguments:

interface_num	The number of the I²C interface of device. By default, a zero interface is used.

Return:

The status of the operation as a standard error code.

Example:

See an example of the i2cExecuteCommands API.

2.41 i2cStopMultipleTransaction

Stops the previously started multiple I²C transaction.

Signature:

```
int32_t i2cStopMultipleTransaction()
```

Return:

The status of the operation as a standard error code.

Example:

See an example of the i2cExecuteCommands API.

2.42 i2cAppendCommand

Appends a short I²C command to internal buffer of the current multiple transaction.

Note: *The size of the multiple transaction is limited by USB packet length which is 64 bytes long. Therefore, the overall size of all commands collected in the transaction's buffer can't be more than 64 bytes. The metadata is reserved for every command in the buffer, decreasing the available size by 4 bytes per appended command.*

Bytes available for the next command are calculated as:

```
AvailBytes = 64 - Sum(payloads) - 4 * (NumCmds + 1) - 2
```

This value is returned by getI2cParam(BytesAvailableForNextCmd).

This limitation applies both for I²C Write and Read requests. The Read response is packed into a single USB packet too, e.g. if you create a transaction with a single Read command you cannot request more than 58 bytes.

Signature:

```
int32_t i2cAppendCommand(
    uint8_t addr,
    uint8_t mode,
    const uint8_t* in_buff_p,
    uint32_t length)
```

API Description

Arguments:

addr	The 7-bit address of the I²C slave device.

mode	The mask bits that define the I²C bus protocol signals. Different masks can be ORed.
	Listed in the CyBridgeI2CMode enum:
	`/* I2C Modes. Passed as the mode argument to the i2cDataTransfer API. Can be ORed together */`
	`enum CyBridgeI2CMode : uint8_t {`
	`/* The current operation is I2C Write */`
	`Write = 0x00,`
	`/* The current operation is I2C Read */`
	`Read = 0x01,`
	`/* Generate the I2C Start condition */`
	`GenStart = 0x02,`
	`/* Generate the I2C Restart condition */`
	`GenReStart = 0x04,`
	`/* Generate the I2C Stop condition */`
	`GenStop = 0x08`
	`};`

in_buff_p	The data to send to the I²C slave device. This parameter is used only for Write operation. • Its size should be equal to or bigger than length param. • For Read operation, this array is disregarded and the buffer pointer can be set to nullptr.
length	For Write operation - the number of bytes to send to the I²C slave device. For Read operation - the number of bytes to be read from slave device.

Return:

The status of the operation as a standard error code.

The CyBridgeNoMem error is returned if internal buffer is not large enough to store the given command. The status of the operation as a standard

Example:

```
const std::vector<uint8_t> write_5b {0x00, 0x02, 0x05, 0x06, 0xBB};
bridge_p->i2cAppendCommand(I2CAddr,
    ModeWrite,
    write_5b.data(),
    write_5b.size());
```

2.43 i2cExecuteCommands

Initiates the execute commands action. This call makes firmware to execute all commands previously collected in the current transaction.

Signature:

```
int32_t i2cExecuteCommands ()
```

Return:

If >= 0 - the number of the actually executed commands, otherwise the status of the operation as a standard error code.

Example:

```
bridge_p->i2cStartMultipleTransaction();
const std::vector<uint8_t> write_buff {0x00, 0x02};

bridge_p->i2cAppendCommand(I2CAddr, Write | GenStart | GenStop, write_buff.data(),
write_buff.size());

int32_t num_commands = bridge_p->i2cExecuteCommands();
if (num_commands < 0) {
    cout << "Failed to execute several I2C commands in a single transaction"
         << endl;
    return;
}

std::vector<uint8_t> resp_buff(64);
bridge_p->i2cGetCmdResponse(0, resp_buff.data(), resp_buff.size());

bridge_p->i2cStopMultipleTransaction();
```

2.44 i2cGetCmdResponse

Returns the response of the I²C command by the given index.

Signature:

```
int32_t i2cGetCmdResponse(
    uint32_t cmd_index,
    uint8_t * resp_buffer_p,
    uint32_t buff_len)
```

Arguments:

cmd_index	The index of the command. This index corresponds to the position of the I²C command being added to the transaction.
resp_buffer_p	The pointer to the buffer where response of the command is returned. • For Read operation – Contains the number of bytes actually read from the device. • For Write operation – Contains the ACK / NACK result for every sent byte. Normally, when the first NACK byte occurs, all others are NACKed as well.
buff_len	The length of the response buffer. *Note: This buffer size must not be less than number of bytes requested by the appropriate command in the i2cAppendCommand() API call.*

Return:

If >= 0 - the number of data bytes written to the response buffer. Otherwise - the status of the operation as the standard error code.

Example:

```
std::vector<uint8_t> resp_buff(64);
bridge_p->i2cGetCmdResponse(0, resp_buff.data(), resp_buff.size());
```

2.45 cancelTransfer

Cancels the (possibly stuck) I²C/SPI data transfer operation. This call aborts only a single pending or subsequent data transfer operation. The call to this method affects only next APIs: ***i2cDataTransfer***(), ***spiDataTransfer***(), ***i2cExecuteCommands***(). As result of invoking this method the aborted transfer operation returns the control flow to the caller with `CyBridgeOperationCancelled` error code.

Signature:

```
int32_t cancelTransfer()
```

Return:

The status of the operation as the standard error code.

Example:

```
bridge_p->cancelTransfer();
```

2.46 gpioSetMode

Sets the Drive mode on KitProg3 GPIO Pin.

Signature:

```
int32_t gpioSetMode(uint8_t pin, CyBridgeGpioMode mode)
```

Arguments:

Pin	Defines the available pin for GPIO Bridging. For CY8CKIT-062S2-43012, CYW9P62S1-43438EVB-01, CYW9P62S1-43012EVB-01, CY8CKIT-064B0S2-4343W, CY8CKIT-064S0S2-4343W, CY8CKIT-040T, and KIT_XMC72_EVK, two KitPog3 pins can be used for GPIO Bridging: • 0x35 – connected to user button 2 in header J21 on the target device • 0x36 - connected to the TP21 on target device For CYW555xx only one KitProg3 pin is available for GPIO Bridging - 3[5] (0x35). Can be defined as one of the values from enum: `enum CyBridgeGpioPinsKp3: uint8_t { gpio_port3_pin5 = 0x35, gpio_port3_pin5 = 0x36 }`
Mode	Can be defined as one of the values from `CyBridgeGpioMode` enum: `/* Drive GPIO Pin modes. */ enum CyBridgeGpioMode : uint8_t { /* PIN_DM_DIG_HIZ */ HiZ = 0x00, /* PIN_DM_RES_UP */ ResUp = 0x01, /* PIN_DM_RES_DWN */ ResDw = 0x02, /* PIN_DM_OD_LO */ OdLo = 0x03, /* PIN_DM_OD_HI */ OdHi = 0x04, /* PIN_DM_STRONG */ DmStr = 0x05, /* PIN_DM_RES_UPDWN */ ResUpdwn = 0x06 };`

Return:

error code.

Example:

```
uint8_t pin = 0x35;
bridge_p->gpioSetMode(pin, CyBridgeGpioMode::DmStr);
```

2.47 gpioSetState

Sets the KitProg3 GPIO pin state.

Signature:

```
int32_t gpioSetState(uint8_t pin, CyBridgeGpioState state)
```

Arguments:

Pin	Defines the available pin for GPIO Bridging. For CY8CKIT-062S2-43012, CYW9P62S1-43438EVB-01, CYW9P62S1-43012EVB-01, CY8CKIT-064B0S2-4343W, CY8CKIT-064S0S2-4343W, CY8CKIT-040T, and KIT_XMC72_EVK, two KitPog3 pins can be used for GPIO Bridging: • 0x35 – connected to user button 2 in the header J21 on the target device • 0x36 - connected to the TP21 on target device For CYW555xx only one KitProg3 pin is available for GPIO Bridging - 3[5] (0x35). Can be defined as one of the values from enum: `enum CyBridgeGpioPinsKp3: uint8_t { gpio_port3_pin5 = 0x35, gpio_port3_pin5 = 0x36 }`
State	Can be defined as one of the values from enum: `/* GPIO Pin State Change. */ enum CyBridgeGpioState : uint8_t { /* Clear */ Clear = 0x00, /* Set */ Set = 0x01 };`

Return:

error code.

Example:

```
uint8_t pin = 0x36;
bridge_p->gpioSetState(pin, CyBridgeGpioState::Set);
```

2.48 gpioRead

Reads current state of the KitProg3 GPIO pin.

Signature:

```
int32_t gpioRead(uint8_t pin, uint8_t &state)
```

Arguments:

pin	Defines the available pin for GPIO Bridging. For CY8CKIT-062S2-43012, CYW9P62S1-43438EVB-01, CYW9P62S1-43012EVB-01, CY8CKIT-064B0S2-4343W, CY8CKIT-064S0S2-4343W, CY8CKIT-040T, and KIT_XMC72_EVK, two KitPog3 pins can be used for GPIO Bridging: • 0x35 – connected to user button 2 in the header J21 on the target device • 0x36 - connected to the TP21 on the target device For CYW555xx only one KitProg3 pin is available for GPIO Bridging - 3[5] (0x35). Can be defined as one of the values from enum: `enum CyBridgeGpioPinsKp3: uint8_t { gpio_port3_pin5 = 0x35, gpio_port3_pin5 = 0x36 }`
state	Returns the current pin state, can be either Low (0) or High (1)

Return:

error code.

Example:

```
uint8_t pin = 0x36;
uint8_t state;
    bridge_p->gpioRead(pin, state);
    if (state == 0x01) {
        std::cout << "\t Signal on pin is 1 (High) " << std::endl;
    }
```

2.49 gpioStateChanged

Returns whether the state of the KitProg3 GPIO pin has changed.

Signature:

```
int32_t gpioStateChanged(uint8_t pin, CyBridgeGpioStateTransition &state)
```

Arguments:

pin	Defines the available pin for GPIO Bridging. For CY8CKIT-062S2-43012, CYW9P62S1-43438EVB-01, CYW9P62S1-43012EVB-01, CY8CKIT-064B0S2-4343W, CY8CKIT-064S0S2-4343W, CY8CKIT-040T, and KIT_XMC72_EVK, two KitPog3 pins can be used for GPIO Bridging: • 0x35 – connected to the user button 2 in header J21 on the target device • 0x36 - connected to the TP21 on the target device For CYW555xx only one KitProg3 pin is available for GPIO Bridging - 3[5] (0x35). Can be defined as one of the values from enum: `enum CyBridgeGpioPinsKp3: uint8_t { gpio_port3_pin5 = 0x35, gpio_port3_pin5 = 0x36 }`
state	Reference to a variable where one of the following values is returned: `/* GPIO Pin States. */ enum CyBridgeGpioStateTransition : uint8_t { /* State has not changed since previous read */ Unchanged = 0x00, /* Transition from low to high occured */ LowToHigh = 0x01, /* Transition from high to low occured */ HighToLow = 0x02 };`

Return:

error code.

Example:

```
uint8_t pin = 0x36;
CyBridgeGpioStateTransition stateChange = 0;
    ret = bridge_p->gpioStateChanged(pin, stateChange);
    if (stateChange == CyBridgeGpioStateTransition::LowToHigh) {
        std::cout << "\n >> State Transition from Low to High occured "
                  << std::endl;
    }
```

2.50 resetDevice

Issues the KitProg3-based device reset request.

Signature:

```
int32_t resetDevice()
```

Return:

error code.

Example:

```
bridge_p->resetDevice();
```

Limitation:

On Windows 7 host machines, after resetDevice API KitProg3 COM port interface is not recognized in the system. This issue can be observed due to system demanding longer time for the device to be disconnected, in order driver to be successfully disabled and reenabled, while the KitProg3-based device is being almost instantaneously reconnected after reset. There are two ways to solve this problem:

1. Manually re-enable COM port device in the Device Manager, or
2. Issue the second resetDevice API.

2.51 resetTarget

Issues the target MCU reset request.

Signature:

```
int32_t resetTarget()
```

Return:

error code.

Example:

```
bridge_p->resetTarget();
```

2.52 setUartFlowControl

Sets the UART flow control mode for the requested KitProg3 UART .

Signature:

```
int32_t setUartFlowControl(cyBridgeKitProgUartPortNumber port_number,
cyBridgeKitProgUartFlowControl mode)
```

Arguments:

port_number	Defines the KitProg3 UART port to configure. Can be defined as one of the values from enum:
	`enum cyBridgeKitProgUartPortNumber : uint8_t {`
	`KitProg3PrimaryUart = 0x00,`
	`KitProg3SecondaryUart = 0x01`
	`}`

Mode	Value of UART flow control mode to set for the chosen KitProg3 UART:
	`/* The supported values for KitProg3 UART flow control */`
	`enum cyBridgeKitProgUartFlowControl : uint8_t {`
	`/* Unset UART HW flow control */`
	`NoneMode = 0x00,`
	`/* Set UART HW flow control */`
	`HardwareMode = 0x01,`
	`};`

Return:

error code.

Example:

```
cyBridgeKitProgUartPortNumber port = cyBridgeKitProgUartPortNumber::KitProg3PrimaryUart;
cyBridgeKitProgUartFlowControl mode = cyBridgeKitProgUartFlowControl::HardwareMode;
ret = bridge_p->setUartFlowControl(port, mode);
if (ret < 0) {
     std::cout << "\n >> >> UART flow control was not set to hardware." << std::endl;
}
```

2.53 getUartFlowControl

Retrieves the UART flow control mode set for requested KitProg3 UART

Signature:

```
int32_t getUartFlowControl(cyBridgeKitProgUartPortNumber port_number,
cyBridgeKitProgUartFlowControl& mode)
```

Arguments:

port_number	Defines the KitProg3 UART port to configure. Can be defined as one of the values from enum:
	`enum cyBridgeKitProgUartPortNumber : uint8_t {`
	`  KitProg3PrimaryUart = 0x00,`
	`  KitProg3SecondaryUart = 0x01`
	`}`

Mode	Reference to a variable where one of the following values is returned:
	`/* The supported values for KitProg3 UART flow control */`
	`enum cyBridgeKitProgUartFlowControl : uint8_t {`
	`  /* Unset UART HW flow control */`
	`  NoneMode = 0x00,`
	`  /* Set UART HW flow control */`
	`  HardwareMode = 0x01,`
	`};`

Return:

error code.

Example:

```
cyBridgeKitProgUartPortNumber port = cyBridgeKitProgUartPortNumber::KitProg3PrimaryUart;
cyBridgeKitProgUartFlowControl mode = cyBridgeKitProgUartFlowControl::HardwareMode;
ret = bridge_p->getUartFlowControl(port, mode);
if (ret < 0) {
     if (mode == cyBridgeKitProgUartFlowControl::HardwareMode) {
             std::cout << "\n UART flow control is set to hardware." << std::endl;
     }
}
```

2.54 initRtt

Initializes the RTT interface.

Signature:

```
int32_t initRtt()
```

Return:

error code.

Example:

```
ret = bridge_p->initRtt();
if (ret == CyBridgeErrors::CyBridgeNoError) {
    std::cout << "\n The RTT interface has been created." << std::endl;
}
```

2.55 cleanRtt

Destroys RTT interface and releases all resources.

Signature:

```
int32_t cleanRtt()
```

Return:

error code.

Example:

```
ret = bridge_p->cleanRtt();
if (ret == CyBridgeErrors::CyBridgeNoError) {
     std::cout << "\n The RTT interface was destroyed." << std::endl;
}
```

2.56 rttConnectSocketToHost

Attempts to make a connection to the host on a given port

Signature:

```
int32_t rttConnectSocketToHost(const char* hostName, uint16_t port)
```

Arguments:

Hostname	Specifies host to connect to, may be an IP address in string form (e.g., "43.195.83.32"), or it may be a host name (e.g., "example.com").
Port	Specifies a TCP port to connect to.

Return:

error code.

Example:

```
const char* hostName = "127.0.0.1";
uint16_t port = 50567;
int32_t ret = bridge_p->rttConnectSocketToHost(hostName, port);
if (ret == CyBridgeErrors::CyBridgeNoError) {
    std::cout << "\n The connection to the " << hostName << " on the " << port << " was
established." << std::endl;
}
```

2.57 rttDisconnect

Closes a connection made to the host on a given port

Signature:

```
int32_t rttDisconnect()
```

Return:

error code.

Example:

```
ret = bridge_p->rttDisconnect();
if (ret == CyBridgeErrors::CyBridgeNoError) {
    std::cout << "\n Socket was disconnected successfully." << std::endl;
}
```

2.58 rttRead

Reads the data available on the RTT interface. This is a blocking API that does not finish until a timeout or the number of expected bytes is received.

Signature:

```
int32_t rttRead(uint8_t* buff_p, uint32_t buff_len, uint32_t timeout_ms = DefaultTimeout)
```

Arguments:

buff_p	The pointer to the Read buffer.
buff_len	The number of bytes to read, must be <= of the size of the buff_p buffer.
timeout_ms	The timeout value for operation. The default value is 1000 ms.

Return:

Number of read bytes or error code.

Example:

```
uint8_t readBuff[10000];
uint32_t timeout = 10000;
const uint32_t packet_size = 0x100u;

ret = bridge_p->rttRead(readBuff, packet_size, timeout);
if (ret != packet_size) {
    std::cout << "Number of bytes read " << ret << " with expected "
              << packet_size << "." << std::endl;
 }
```

2.59 rttWrite

Sends a raw data over RTT interface

Signature:

```
int32_t rttWrite(const uint8_t* buff_p, uint32_t buff_len)
```

Arguments:

buff_p	The pointer to the data to send.
buff_len	The number of the bytes to send.

Return:

error code.

Example:

```
uint8_t writeBuff[5] = {0x01, 0x02, 0x03, 0x04, 0x05};
uint32_t num_of_bytes = 5;

ret = bridge_p->rttWrite(writeBuff, num_of_bytes);
if (ret == CyBridgeErrors::CyBridgeNoError) {
    std::cout << "\n Write Operation was successful." << std::endl;
}
```

2.60 rttIsActive

Checks if connection between socket and host has been established

Signature:

```
int32_t rttIsActive()
```

Return:

Bool value or error code.

Example:

```
ret = bridge_p->rttIsActive();
if (ret < CyBridgeErrors::CyBridgeNoError) {
    std::cout << "\n An error has occurred: " << bridge_p->getErrorDescription(ret) <<
std::endl;
} else {
    std::cout << "\n The socket to host connection was " << ((ret == 0) ? " not
established." : " established.") << std::endl;
}
```

2.61 rttSetReadyReadEnable

Switches on/off RTT Read Data Mode. Should be called before rttRead

Signature:

```
int32_t rttSetReadyReadEnable(bool readyReadEnable)
```

Arguments:

readyReadEnable	Defines the RTT mode, if true, new data available for reading from the device's current read channel will be read and added to local buffer.

Return:

error code.

Example:

```
res = bridge_p->rttSetReadyReadEnable(true);
```

2.62 rttClearData

Clears data in RTT interface Read and Write buffers, discards all information available for reading

Signature:

```
int32_t rttClearData()
```

Return:

error code.

Example:

```
ret = bridge_p->rttClearData();
if (ret == CyBridgeErrors::CyBridgeNoError) {
    std::cout << "\n RTT Buffers were cleared successfully. << std::endl;
} else {
    std::cout << "\n An error has occurred: " << bridge_p->getErrorDescription(ret) <<
std::endl;
}
```

3 Rules of CyBridge Use

3.1 Multithreading

- The `CreateCyBridge`(), `initialize`(), and `exit`()/`delete`() functions should be performed on the same thread.
- The `openDevice`() and `closeDevice`() functions should be performed on the same thread.
- If a device is opened on a non-create thread, then `closeDevice`() should be called before releasing CyBridge.
- Communication with devices can be performed on a non-create/open thread; do not use multiple threads to communicate with the same device.
- If communication with the device is performed on a non `openDevice`() thread, then all communication with the device must be stopped before calling `closeDevice`()/`openDevice`() APIs or releasing CyBridge.
- Simultaneous calls of CyBridge APIs from different threads is not supported.

3.2 Callbacks

- Do not call CyBridge APIs and do not change CyBridge state in callback handlers.
- Do not perform time consuming operations in callback handlers.

3.3 Drivers

KitProg3 devices may require installing a driver on the Windows platform. This driver is part of the fw-loader tool. To download it follow the link: https://github.com/Infineon/Firmware-loader.

3.4 Order of Creation and Destruction

If there are several CyBridge instances, the first created instance must be destroyed by the last one. This limitation is imposed by the initialization order of the internal libusb library.

3.5 RTT Bridging Specifics

The RTT Bridging APIs must not be used by multiple clients/threads at once. This stems from RTT implementation being designed to provide only single-threaded methods.

4 Error and Status Codes

All CyBridge error and status codes are grouped into the `CyBridgeErrors` structure as follows. Each code contains a description.

```
/// \brief CyBridge errors and statuses
enum CyBridgeErrors : int32_t {
  /* The operation completed successfully */
  CyBridgeNoError = 0,
  /* Not enough memory to complete the operation */
  CyBridgeNoMem = -1,
  /* An error of the execution. See the operation logs for details */
  CyBridgeOperationError = -2,
  /* Requested an invalid argument. */
  CyBridgeInvalidArgument = -3,
  /* Requested an unsuppoted parameter value. */
  CyBridgeUnsupportedValue = -4,
  /* The specified device was not found. */
  CyBridgeDeviceNotFound = -5,
  /* Requested an unsupported interface number.
  CyBridgeNotSupportedInterfaceNumber = -6,
  /* The device is not opened. */
  CyBridgeDeviceNotOpened = -7,
  /* The device does not have a power-monitor interface. */
  CyBridgeNoPowerMonitor = -8,
  /* The device does not have a power-control interface. */
  CyBridgeNoPowerControl = -9,
  /* The device does not have an SWD interface. */
  CyBridgeNoSwdBridge = -10,
  /* The device does not have an SPI interface. */
  CyBridgeNoSpiBridge = -11,
  /* The device does not have an I2C interface. */
  CyBridgeNoI2cBridge = -12,
  /* The requested operation is not supported. */
  CyBridgeNotSupported = -13,
  /* The size of the output buffer is not enough to contain the whole result. */
  CyBridgeNotEnoughOutBuffer = -14,
  /* The device is already opened by either another instance of the CyBridge library or by
another application. */
  CyBridgeDeviceInUse = -15,
  /* The "initialize API" was not called. */
  CyBridgeNotInitialized = -16,
  /* The operation has timed out. */
  CyBridgeOperationTimeout = -17,
  /* The FW update is forbidden. Returned by upgradeFw() only. */
  CyBridgeFwUpdateForbidden = -18,
  /* Access denied. Check the access permissions for the device. */
  CyBridgeAccessDenied = -19,
  /* The device does not have a JTAG interface. */
  CyBridgeNoJtagBridge = -20,
  /* The device does not have a UART interface. */
  CyBridgeNoUartBridge = -21,
  /* The operation is aborted due to a timeout. */
  CyBridgeOperationIsTimedOut = -22,
  /* The FW must be upgraded. Returned by openDevice() only */
  CyBridgeNeedFWUpgrade = -23,
  /* The CyBridge must be restarted to support the device. Returned by openDevice() only */
  CyBridgeNeedRestart = -24,
  /* The CyBridge failed to set the requested voltage. Returned by setVoltage() only */
  CyBridgeFailToSetVoltage = -25,
  /* The device does not have a USB interface. */
  CyBridgeNoUsbBridge = -26,
  /* The long data transfer operation was canceled. */
  CyBridgeOperationCancelled = -27,
  /* The requested UART baud rate value is not supported. */
  CyBridgeUnsupportedUARTBaudRate = -28,
  /* The requested value of UART data bits is not supported.
  CyBridgeUnsupportedUARTDataBits = -29,
```

Error and Status Codes

```
    /* The requested value of UART stop bits is not supported. */
    CyBridgeUnsupportedUARTStopBits = -30,
    /* The requested UART parity value is not supported. */
    CyBridgeUnsupportedUARTParity = -31,
    /* The device does not have a GPIO interface. */
    CyBridgeNoGpioBridge = -32,
    /* No FW path provided */
    CyBridgeNoFwProvided = -33,
    /* The 'initialize API for RTT' was not called */
    CyBridgeRttNotInitialized = -34,
    /* Rtt socket connection to host failed */
    CyBridgeRttConnectionFailed = -35,
    /* Rtt socket already connected */
    CyBridgeRttAlreadyConnected = -36,
    /* Rtt socket not connected to host */
    CyBridgeRttNotConnected = -37,
};
```

5 Examples

5.1 Display Connected Hardware

```
#include <iostream>
#include "cybridge.h"

int main() {
  // Create a CyBridge instance
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge | CyBridgeDevices::UART);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    // Get a list of the connected devices
    CyBridgeDevice** dev_list = nullptr;
    ret = bridge_p->getDeviceList(&dev_list);
    if (ret >= 0) {
      std::cout << "The number of the connected devices = " << ret << std::endl;
      for (int i = 0; i < ret; i++) {
        std::cout << "\nN = " << i + 1 << " Name = " << dev_list[i]->Name
                  << ", Short Name = " << dev_list[i]->ShortName
                  << ", vid = 0x" << std::hex << dev_list[i]->Vid
                  << ", pid = 0x" << dev_list[i]->Pid << std::dec
                  << ", SN = " << dev_list[i]->SerialNumber
                  << ", Manufacturer = " << dev_list[i]->Manufacturer
                  << ", Product = " << dev_list[i]->Product
                  << ", Location = " << dev_list[i]->Location
                  << ", Probe name = " << dev_list[i]->ProbeName
                  << ", Communication = " << dev_list[i]->ComType << std::endl;
      }
      bridge_p->releaseDeviceList(dev_list);
    } else {
      std::cout << "Cannot get the list of the connected devices. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }
  } else {
    std::cout << "Cannot initialize the CyBridge. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
  }

  // Release all resources
  bridge_p->exit();

  return 0;
}
```

Examples

Output:

```
The number of connected devices = 3.

N = 1 Name = KitProg3-0B0B0F9701047400, Short Name = KP3-0B0B0F9701047400, vid = 0x4b4, pid
= 0xf154, SN = 0B0B0F9701047400, Manufacturer = Cypress Semiconductor, Product = KitProg3
bridge, Location = \\?\hid#vid_04b4&pid_f154&mi_01#a&1fd8807f&0&0000#{4d1e55b2-f16f-11cf-
88cb-001111000030}, Supported kits = CY8CKIT_062_BLE,CY8CKIT_062_WiFi_BT,CYW943012P6EVB-01,
Probe name = KitProg3, Communication = hid

N = 2 Name = COM3, Short Name = COM3, vid = 0x8086, pid = 0x1e3d, SN = , Manufacturer =
Intel, Product = Intel(R) Active Management Technology - SOL, Location = COM3, Probe name =
, Communication = serial

N = 3 Name = COM29, Short Name = COM29, vid = 0x4b4, pid = 0xf154, SN = 0B0B0F9701047400,
Manufacturer = Microsoft, Product = USB Serial Device, Location = COM29, Probe name = ,
Communication = serial
```

5.2 Using Connect/Disconnect Events

```
#include <iostream>
#include "cybridge.h"

//
// connectEvent() -
//
// Arguments:
//      dev_p - The pointer to the CyBridgeDevice structure with a description of the
connected device.
//      is_connected - "true" if the device is connected, "false" - if disconnected.
//      client_p - The user-supplied data to the setConnectCallback API.
//
// A user's registered callback for the device Connect/Disconnect events.
//
void connectEvent(CyBridgeDevice* dev_p, bool is_connected, void* /*client_p*/) {
  if (is_connected) {
    std::cout << "Connected: ";
  } else {
    std::cout << "Disconnected: ";
  }

  std::cout << "Name = " << dev_p->Name << " Short Name = " << dev_p->ShortName
            << ", vid = 0x" << std::hex << dev_p->Vid << ", pid = 0x"
            << dev_p->Pid << std::dec << ", SN = " << dev_p->SerialNumber
            << ", Manufacturer = " << dev_p->Manufacturer << ", Product = "
            << dev_p->Product << ", Location = " << dev_p->Location
            << ", Probe name = "  << dev_p->ProbeName << ", Communication = "
            << dev_p->ComType << ", Operation speed = " << dev_p->OperationSpeed
            << "\n" << std::endl;
}

int main() {
  // Create a CyBridge instance.
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Register the connect callbacks.
  bridge_p->setConnectCallback(&connectEvent, nullptr);

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge | CyBridgeDevices::UART);
  if (CyBridgeErrors::CyBridgeNoError != ret) {
    std::cout << "Cannot initialize CyBridge. Err = " << bridge_p->getErrorDescription(ret)
<< std::endl;
  }

  // Release all resources
  bridge_p->exit();

  return 0;
}
```

Examples

Output:

```
Connected: Name = KitProg3-0B0B0F9701047400 Short Name = KP3-0B0B0F9701047400, vid = 0x4b4,
pid = 0xf154, SN = 0B0B0F9701047400, Manufacturer = Cypress Semiconductor, Product =
KitProg3 bridge, Location = \\?\hid#vid_04b4&pid_f154&mi_01#a&1fd8807f&0&0000#{4d1e55b2-
f16f-11cf-88cb-001111000030}, Supported kits =
CY8CKIT_062_BLE,CY8CKIT_062_WiFi_BT,CYW943012P6EVB-01, Probe name = KitProg3, Communication
= hid, Operation speed = 0

Connected: Name = COM3 Short Name = COM3, vid = 0x8086, pid = 0x1e3d, SN = , Manufacturer =
Intel, Product = Intel(R) Active Management Technology - SOL, Location = COM3, Probe name =
, Communication = serial, Operation speed = 0

Connected: Name = COM29 Short Name = COM29, vid = 0x4b4, pid = 0xf154, SN =
0B0B0F9701047400, Manufacturer = Microsoft, Product = USB Serial Device, Location = COM29,
Probe name = , Communication = serial, Operation speed = 0
```

5.3 Enumerating Supported Interfaces

```
#include <iostream>
#include "cybridge.h"

int main() {
  // Create a CyBridge instance.
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge | CyBridgeDevices::UART);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    // Get the list of the connected devices.
    CyBridgeDevice** dev_list = nullptr;
    int count = bridge_p->getDeviceList(&dev_list);
    if (count >= 0) {
      std::cout << "Number of connected devices = " << count << std::endl;
      for (int i = 0; i < count; i++) {
        std::cout << "\nN = " << i + 1 << " Name = " << dev_list[i]->Name << std::endl;
        ret = bridge_p->openDevice(dev_list[i]->Name);
        if (ret < 0) {
          std::cout << "\t Cannot open device '" << dev_list[i]->Name << "'. Err = "
                    << bridge_p->getErrorDescription(ret) << std::endl;
          continue;
        } else if (ret == CyBridgeErrors::CyBridgeNeedFWUpgrade) {
          std::cout << "\t Recommended to upgrade Firmware of '" << dev_list[i]->Name
                    << "'" << std::endl;
        }

        CyBridgeDeviceProps* props;
        ret = bridge_p->getDeviceProps(&props);
        if (ret < 0) {
          std::cout << "\t Cannot get device '" << dev_list[i]->Name
                    << "' props. Err = " << ret << std::endl;
        } else {
          std::cout << "\t    The device props: "
                    << "Num of interfaces = " << (int)(props->NumIfs)
                    << ", Interfaces = " << props->ListIfs << std::endl;
        }
        bridge_p->closeDevice();
      }
      bridge_p->releaseDeviceList(dev_list);
    } else {
      std::cout << "Cannot get list of connected devices. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }
  } else {
    std::cout << "Cannot initialize CyBridge. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
  }

  // Release all resources
  bridge_p->exit();

  return 0;
}
```

Examples

Output:

```
The number of connected devices = 3.

N = 1 Name = KitProg3-0B0B0F9701047400
             The device props: Num of interfaces = 4, Interfaces =
i2c,spi,power_monitor,power_control

N = 2 Name = COM3
             The device props: Num of interfaces = 1, Interfaces = uart

N = 3 Name = COM29
             The device props: Num of interfaces = 1, Interfaces = uart
```

5.4 I²C Example

This example demonstrates the I²C Read and Write operations.

```
#include <iostream>
#include <vector>
#include <sstream>
#include <iomanip>
#include "cybridge.h"

const char I2cSupportedDevice[] = "KitProg3-0B0B0F9701047400";
const size_t BufferSize = 500;
char buff[BufferSize];
const uint8_t I2CAddr = 8;

int main() {
  // Create a CyBridge instance
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    do {
      ret = bridge_p->openDevice(I2cSupportedDevice);
      if (ret < 0) {
        std::cout << "Cannot open device '" << I2cSupportedDevice << "'. Err = "
                  << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }
      ret = bridge_p->getI2cParam(CyBridgeI2cGetParams::I2cFreqListHz, buff,
                                  BufferSize);
      if (ret < 0) {
        std::cout << "Error of retrieving 'CyBridgeI2cGetParams::I2cFreqListHz' "
                     "parameter. Err = " << bridge_p->getErrorDescription(ret)
                  << std::endl;
        break;
      }

      std::cout << "I2C Master: Supported frequencies, Hz = " << buff << std::endl;

      // Set the I2C clock to 100 kHz
      ret = bridge_p->setI2cParam(CyBridgeI2cSetParams::I2cFreqHz, 100000);

      // The I2C Write
      uint32_t num_bytes_to_write = 8;
      std::vector<uint8_t> write_buff = {1, 2, 3, 4, 5, 6, 7, 8};
      uint32_t response_buff_size = num_bytes_to_write + 1;
      std::vector<uint8_t> response_buff(response_buff_size);

      ret = bridge_p->i2cDataTransfer(I2CAddr,
        CyBridgeI2CMode::Write | CyBridgeI2CMode::GenStart | CyBridgeI2CMode::GenStop,
        0, &write_buff[0], num_bytes_to_write, &response_buff[0], response_buff_size);
      if (ret < 0) {
        std::cout << "\t Cannot perform I2C write on '" << I2cSupportedDevice
                  << " device'. Err = " << bridge_p->getErrorDescription(ret)
                  << std::endl;
        break;
      }

      std::stringstream print_str;
      for (size_t index = 0; index < num_bytes_to_write; index++) {
        print_str << std::hex << std::setfill('0') << std::setw(2) << std::uppercase
                  << static_cast<int>(write_buff[index])
                  << (response_buff[index + 1] ? "+" : "-") << " ";
      }
```

Examples

```
        std::cout << "Addr " << (response_buff[0] ? "ACK" : "NACK") << ", Write data = "
                  << print_str.str() << std::endl;

        // The I2C Read
        uint32_t num_bytes_to_read = 8;
        response_buff_size = num_bytes_to_read + 1;
        response_buff.resize(response_buff_size);

        ret = bridge_p->i2cDataTransfer(I2CAddr,
          CyBridgeI2CMode::Read | CyBridgeI2CMode::GenStart | CyBridgeI2CMode::GenStop,
          num_bytes_to_read, nullptr, 0, &response_buff[0], response_buff_size);

        if (ret < 0) {
          std::cout << "\t Cannot perform I2C read on '" << I2cSupportedDevice
                    << " device'. Err = " << bridge_p->getErrorDescription(ret)
                    << std::endl;
          break;
        }

        print_str.str("");
        for (size_t index = 0; index < num_bytes_to_read; index++) {
          print_str << std::hex << std::setfill('0') << std::setw(2) << std::uppercase
                    << static_cast<int>(response_buff[index + 1]) << " ";
        }

        std::cout << "Addr " << (response_buff[0] ? "ACK" : "NACK") << ", Read data = "
                  << print_str.str() << std::endl;

        bridge_p->closeDevice();
      } while (0);

    } else {
      std::cout << "Cannot initialize CyBridge. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }

    // Release all resources
    bridge_p->exit();

    return 0;
}
```

Output:

The Read data depends on the I²C Slave operation:

```
I2C Master: Supported frequencies, Hz = 50000,100000,400000,1000000
Addr ACK, Write data = 01+ 02+ 03+ 04+ 05+ 06+ 07+ 08+
Addr ACK, Read data = 00 02 03 04 05 06 07 08
```

5.5 Multiple I²C Transactions Example

The multiple I²C transactions API is the extension to the existing I²C communication API. It allows a client to pack several short I²C commands into a single transaction and send them with one USB transfer. This benefits in increased communication performance over I²C protocol. However, this API is well suited only for a limited sequence of short I²C data transfers.

This example below packs several Write and one Read commands into a single I²C transaction, then executes it and displays responses for every executed command.

```
#include <iostream>
#include <vector>
#include <sstream>
#include <iomanip>
#include "cybridge.h"

const char I2cSupportedDevice[] = "KitProg3-0B0B0F9701047400";
const size_t BufferSize = 500;
char buff[BufferSize];
const uint8_t I2CAddr = 8;

int main() {
  const std::vector<uint8_t> write_5b {0x00, 0x02, 0x05, 0x06, 0xBB};
  const std::vector<uint8_t> write_6b {0x01, 0x03, 0x00, 0x06, 0x04, 0xAA};
  bool opened = false;

  // Create a CyBridge instance
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    do {
      ret = bridge_p->openDevice(I2cSupportedDevice);
      if (ret < 0) {
        std::cout << "Cannot open device '" << I2cSupportedDevice << "'. Err = "
                  << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }
      opened = true;

      ret = bridge_p->
       getI2cParam(CyBridgeI2cGetParams::IsMultipleTransactionSupported, buff,
                   BufferSize);
      if (ret < 0) {
        std::cout << "Error of retrieving 'IsMultipleTransactionSupported' "
                  << "parameter. Err = " << bridge_p->getErrorDescription(ret)
                  << std::endl;
        break;
      }

      if (buff != std::string("true")) {
        std::cout << "i2c Multiple Transaction mode is not supported by device" <<
            std::endl;
      }

     bridge_p->i2cStartMultipleTransaction();

      // Write 5 bytes
      ret = bridge_p->i2cAppendCommand(I2CAddr, Write|GenStart|GenStop,
              write_5b.data(), write_5b.size());
      if (ret != CyBridgeErrors::CyBridgeNoError) break;
```

Examples

```
        // Read 20 bytes
        ret = bridge_p->i2cAppendCommand(I2CAddr, Read|GenStart|GenStop, nullptr, 20);
        if (ret != CyBridgeErrors::CyBridgeNoError) break;

        // Write 6 bytes
        ret = bridge_p->i2cAppendCommand(I2CAddr, Write|GenStart|GenStop,
                 write_6b.data(), write_6b.size());
        if (ret != CyBridgeErrors::CyBridgeNoError) break;

        auto num_commands = bridge_p->i2cExecuteCommands();
        if (num_commands < 0) {
           std::cout << "Error occured while executing I2C commands in a single" <<
                     "transaction" << std::endl;
           ret= num_commands;
           break;
        }
        std::cout << "Executed " << num_commands << " commands in a multiple I2C" <<
             "transaction" << std::endl;

        std::vector<uint8_t> resp_buffer(64);
        for (int i =0; i<num_commands; ++i) {
           auto resp_size = bridge_p->i2cGetCmdResponse(static_cast<uint32_t>(i),
                resp_buffer.data(), resp_buffer.size());

           std::cout << "response [" << i << "]: ";
           for (int index = 0; index < resp_size; index++) {
             std::cout << std::hex << std::setfill('0') << std::setw(2) <<
                       std::uppercase << static_cast<int>(resp_buffer[index]);
           }
           std::cout << std::endl;
        }
        bridge_p->i2cStopMultipleTransaction();

      } while (0);

      if (opened) bridge_p->closeDevice();

    } else {
      std::cout << "Cannot initialize CyBridge. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }

    // Release all resources
    bridge_p->exit();

    return 0;
}
```

Output:

The Read data depends on the I²C Slave operation:

```
Executed 3 commands in a multiple I2Ctransaction
response [0]: 0101010101
response [1]: F0020506BB000000000000000000000000000000
response [2]: 010101010101
```

5.6 SPI Example

```
#include <iomanip>
#include <iostream>
#include <sstream>
#include <vector>
#include "cybridge.h"

const char SpiSupportedDevice[] = "KitProg3-0B0B0F9701047400";
const size_t BufferSize = 500;
char buff[BufferSize];

int main() {
  // Create a CyBridge instance
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    do {
      ret = bridge_p->openDevice(SpiSupportedDevice);
      if (ret < 0) {
        std::cout << "Cannot open device '" << SpiSupportedDevice << "'. Err = "
                  << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }

      uint32_t spi_freq = 120000;

      std::cout << "Setting SPI Frequency to " << spi_freq << " Hz" << std::endl;

      ret = bridge_p->setSpiParam(CyBridgeSpiSetParams::SpiFreqHz, spi_freq);
      if (ret < 0) {
        std::cout << "Error of setting 'CyBridgeSpiSetParams::SpiFreqHz' parameter. "
                     "Err = " << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }

      ret = bridge_p->getSpiParam(CyBridgeSpiGetParams::CurrentSpiFreqHz, buff,
                                  BufferSize);
      if (ret < 0) {
        std::cout << "Error of retrieving 'CyBridgeSpiGetParams::CurrentSpiFreqHz' "
                     "parameter. Err = " << bridge_p->getErrorDescription(ret)
                  << std::endl;
        break;
      }
      std::cout << "Current SPI Frequency, Hz = " << buff << std::endl;

      uint8_t num_bytes_to_transfer = 8;

      std::vector<uint8_t> write_buff(num_bytes_to_transfer);
      std::vector<uint8_t> read_buff(num_bytes_to_transfer);

      for (uint8_t index = 0; index < num_bytes_to_transfer; index++)
          write_buff[index] = index;

      ret = bridge_p->spiDataTransfer(
        CyBridgeSPIMode::AassertSSOnStart | CyBridgeSPIMode::DeassertSSOnStop,
        &write_buff[0], &read_buff[0], num_bytes_to_transfer);
      if (ret < 0) {
        std::cout << "Error of spiDataTransfer. Err = "
                  << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }
```

Examples

```
        std::stringstream print_str;
        for (size_t index = 0; index < num_bytes_to_transfer; index++) {
          print_str << std::hex << std::setfill('0') << std::setw(2)
                    << std::uppercase << static_cast<int>(write_buff[index]) << " ";
        }

        std::cout << "SPI Sent data = " << print_str.str() << std::endl;

        print_str.str("");
        for (size_t index = 0; index < num_bytes_to_transfer; index++) {
          print_str << std::hex << std::setfill('0') << std::setw(2)
                    << std::uppercase << static_cast<int>(read_buff[index]) << " ";
        }

        std::cout << "SPI Read data = " << print_str.str() << std::endl;

        bridge_p->closeDevice();
      } while (0);

    } else {
      std::cout << "Cannot initialize CyBridge. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }

    // Release all resources
    bridge_p->exit();

    return 0;
}
```

Output:

The Read data depends on the SPI Slave operation:

```
Setting SPI Frequency to 120000 Hz
Current SPI Frequency, Hz = 120754
SPI Sent data = 00 01 02 03 04 05 06 07
SPI Read data = 05 06 07 08 09 0A 0B 0C
```

5.7 UART Example

This example shows how to use all available UART Read functions: asynchronous, synchronous Read single packet and buffer.

```
#include <iomanip>
#include <iostream>
#include <sstream>
#include <vector>
#include <thread>
#include "cybridge.h"

const char UartDevice[] = "COM13";

//
// uint8ToHexStrm() -
//
// Arguments:
//      data_p - The pointer to the data to convert.
//      num_bytes_to_convert - The number of bytes to convert.
//      out_strm - The stringstream buffer to contain the conversion result.
//
// Converts uint8_t buffer to stringstream buffer
//
void uint8ToHexStrm(const uint8_t* data_p, size_t num_bytes_to_convert, std::stringstream&
out_strm) {
  out_strm.str("");
  for (uint32_t index = 0; index < num_bytes_to_convert; index++) {
    out_strm << std::hex << std::setfill('0') << std::setw(2) << std::uppercase
             << static_cast<int>(data_p[index]) << " ";
  }
}

//
// uartRxCallback() -
//
// Arguments:
//      data_p - The pointer to the received  data.
//      num_bytes - The number of received bytes in the data_p buffer.
//      client_p - The user-supplied data to setUartRxCallback API.
//
// A user's registered callback for displaying the UART RX logs.
//
void uartRxCallback(uint8_t const* data_p, uint32_t num_bytes, void* /*client_p*/) {
  std::stringstream print_str;
  uint8ToHexStrm(data_p, num_bytes, print_str);
  std::cout << "\t Uart Read data = " << print_str.str() << std::endl;
}

int main() {
  // Create a CyBridge.
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::UART);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    do {
      ret = bridge_p->openDevice(UartDevice);
      if (ret < 0) {
        std::cout << "Cannot open device '" << UartDevice << "'. Err = "
                  << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }

      uint32_t baudRate = 115200;
```

Examples

```
ret = bridge_p->setUartParam(CyBridgeUartSetParams::BaudRate, baudRate);
if (ret < 0) {
  std::cout << "Error of setting 'CyBridgeUartSetParams::BaudRate' parameter. "
               "Err = " << bridge_p->getErrorDescription(ret) << std::endl;
  break;
}
std::cout << "BaudRate set to " << baudRate << std::endl;

std::vector<uint8_t> uart_write_data{0x30, 0x31, 0x32, 0x20};

// The UART Write
ret = bridge_p->uartWrite(&uart_write_data[0],
        static_cast<uint32_t>(uart_write_data.size()));
if (CyBridgeErrors::CyBridgeNoError == ret) {
  std::stringstream print_str;
  uint8ToHexStrm(&uart_write_data[0], uart_write_data.size(), print_str);
  std::cout << "Write data = " << print_str.str() << std::endl;
} else {
  std::cout << "Error of UART Write data. Err = "
            << bridge_p->getErrorDescription(ret) << std::endl;
}

std::vector<uint8_t> uart_read_data(10);

// The UART Read
uint32_t read_timeout = 2000;
std::cout << "\nWaiting " << read_timeout << " ms for " << uart_read_data.size()
          << " bytes of Rx data... " << std::endl;
ret = bridge_p->uartRead(&uart_read_data[0],
        static_cast<uint32_t>(uart_read_data.size()), read_timeout);
if (ret >= 0) {
  std::stringstream print_str;
  uint8ToHexStrm(&uart_read_data[0], uart_read_data.size(), print_str);
  std::cout << "Read data = " << print_str.str() << std::endl;
} else {
  std::cout << "Error of UART Read. Err = "
            << bridge_p->getErrorDescription(ret) << std::endl;
  if (ret != CyBridgeErrors::CyBridgeOperationIsTimedOut) break;
}

// The UART Read Async
read_timeout = 5;
std::cout << "\nStarting the async UART read for "
          << read_timeout << " seconds" << std::endl;
bridge_p->uartReadAsync(&uartRxCallback, &uart_read_data[0],
        static_cast<uint32_t>(uart_read_data.size()), nullptr);

std::this_thread::sleep_for(std::chrono::seconds(read_timeout));

std::cout << "Stopping async UART read " << std::endl;
bridge_p->uartReadAsync(nullptr, nullptr, 0, nullptr);

// UART Read Single Packet

read_timeout = 2000;
std::cout << "\nWaiting " << read_timeout << " ms for Rx data... "
          << std::endl;
ret = bridge_p->uartReadSinglePacket(&uart_read_data[0],
        static_cast<uint32_t>(uart_read_data.size()), read_timeout);

if (ret >= 0) {
  std::stringstream print_str;
  uint8ToHexStrm(&uart_read_data[0], static_cast<size_t>(ret), print_str);
  std::cout << "Read data = " << print_str.str() << std::endl;
} else {
  std::cout << "Error of UART Read. Err = "
            << bridge_p->getErrorDescription(ret) << std::endl;
  if (ret != CyBridgeErrors::CyBridgeOperationIsTimedOut) break;
}
```

Examples

```
        bridge_p->closeDevice();
      } while (0);

    } else {
      std::cout << "Cannot initialize CyBridge. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }

    // Release all resources
    bridge_p->exit();

    return 0;
}
```

Output:

A possible output, depends on the UART send data:

```
BaudRate set to 115200
Write data = 30 31 32 20

Waiting 2000 ms for 10 bytes of Rx data...
Read data = 31 32 33 34 35 36 37 38 39 30

Starting the Async UART Read for 5 seconds:
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
         Uart Read data = 31 32 33 34 35 36 37 38 39 30
Stopping the Async UART Read

Waiting 2000 ms for Rx data...
Read data = 31 32 33 34 35 36 37 38 39 30
```

5.8 Power Control

```
#include <iostream>
#include <thread>
#include <string>
#include <vector>
#include "cybridge.h"

const char DeviceName[] = "KitProg3-0B0B0F9701047400";
const size_t BufferSize = 500;
char buff[BufferSize];

//
// setMeasureVoltage() -
//
// Arguments:
//      bridge_p - The pointer to the CyBridge used for communication with the device.
//      voltage - The volatge to be set.
//
// Shows how to set and measure the voltage.
//
void setMeasureVoltage(CyBridge* bridge_p, uint32_t voltage) {
  std::cout << "\t Set voltage = " << voltage << " mV" << std::endl;
  int ret = bridge_p->setVoltage(voltage);
  if (ret < 0) {
    std::cout << "\t Cannot set voltage " << voltage << " mV. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
    return;
  }

  int32_t measured_voltage;
  bridge_p->getVoltage(&measured_voltage);
  std::cout << "\t\t Measured voltage = " << measured_voltage << " mV";

  char buff[BufferSize];

  ret = bridge_p->getPowerControlParam(CyBridgePowerControlGetParams::LastSetVoltMv,
                                       buff, BufferSize);
  if (ret < 0) {
    std::cout << "\t Error of retrieving "
                 "'CyBridgePowerControlGetParams::LastSetVoltMv' parameter. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
    return;
  }
  std::cout << "\t Set voltage = " << buff << " mV" << std::endl;
}

//
// powerOnOffTest() -
//
// Arguments:
//    bridge_p - The pointer to the CyBridge used for communication with the device.
//
// Shows how to perform the power On/Off.
//
void powerOnOffTest(CyBridge* bridge_p) {
  int ret = bridge_p->powerOn(false);
  if (ret < 0) {
    std::cout << "\t Cannot toggle power Off. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
    return;
  }

  std::this_thread::sleep_for(std::chrono::seconds(1));

  int32_t measured_voltage;
  bridge_p->getVoltage(&measured_voltage);
  std::cout << "\n\t Power Off. Measured voltage = " << measured_voltage
            << " mV";
```

Examples

```
  ret = bridge_p->powerOn(true);
  if (ret < 0) {
    std::cout << "\t Cannot toggle power On. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
    return;
  }

  std::this_thread::sleep_for(std::chrono::seconds(1));

  bridge_p->getVoltage(&measured_voltage);
  std::cout << "\t Power On. Measured voltage = " << measured_voltage
            << " mV" << std::endl;
}

int main() {
  // Create a CyBridge instance.
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    do {
      ret = bridge_p->openDevice(DeviceName);
      if (ret < 0) {
        std::cout << "Cannot open device '" << DeviceName << "'. Err = "
                  << bridge_p->getErrorDescription(ret) << std::endl;
        break;
      }

      bool can_measure_voltage = false;
      bool is_power_ctrl_continuous = false;
      bool is_on_off_supported = false;
      bool is_voltage_control_supported = false;
      std::string discrete_voltages;
      uint32_t min_voltage = 0;
      uint32_t max_voltage = 0;

      ret = bridge_p->getPowerMonitorParam(
             CyBridgePowerMonitorGetParams::CanMeasureVoltage, buff, BufferSize);
      if (ret >= 0) {
        std::cout << "\n    Power Monitor: Can measure the voltage = " << buff;

        if (std::string(buff) == "true") {
          can_measure_voltage = true;
          int32_t voltage;
          bridge_p->getVoltage(&voltage);
          std::cout << ",  Measured voltage = " << voltage << " mV" << std::endl;
        }
      } else {
        std::cout << "\n\n\t Error of retrieving "
                  "'CyBridgePowerMonitorGetParams::CanMeasureVoltage' parameter. Err = "
                  << bridge_p->getErrorDescription(ret) << "\n\n"
                  << std::endl;
      }

      // Check if the power control supports the power On/Off feature
      ret = bridge_p->getPowerControlParam(
              CyBridgePowerControlGetParams::IsOnOffSupported, buff, BufferSize);

      if (ret < 0) {
        std::cout << "\n\n\t Error of retrieving "
                  "'CyBridgePowerControlGetParams::IsOnOffSupported' parameter. Err = "
                  << bridge_p->getErrorDescription(ret) << "\n\n"
                  << std::endl;
```

Examples

```
  break;
}

std::cout << "\n    Power Control: Is On/Off supported = " << buff << std::endl;
if (std::string(buff) == "true") is_on_off_supported = true;

// Check if the power control supports the voltage control.
ret = bridge_p->getPowerControlParam(
      CyBridgePowerControlGetParams::IsVoltCtrlSupported, buff, BufferSize);

if (ret < 0) {
  std::cout << "\n\n\t Error of retrieving "
               "'CyBridgePowerControlGetParams::IsVoltCtrlSupported' parameter. "
               "Err = " << bridge_p->getErrorDescription(ret) << "\n\n"
            << std::endl;
  break;
}

std::cout << "\n    Power Control: Is Voltage Control supported = " << buff
          << std::endl;
if (std::string(buff) == "true") is_voltage_control_supported = true;

// get additional properties of voltage control
if (is_voltage_control_supported) {
  ret = bridge_p->getPowerControlParam(
        CyBridgePowerControlGetParams::IsContinuous, buff, BufferSize);

  if (ret < 0) {
    std::cout << "\n\n\t Error of retrieving "
                 "'CyBridgePowerControlGetParams::IsContinuous' parameter. "
                 "Err = " << bridge_p->getErrorDescription(ret) << "\n\n"
              << std::endl;
    break;
  }

  std::cout << "\n    Power Control: Is continuous = " << buff << std::endl;

  if (std::string(buff) == "true") is_power_ctrl_continuous = true;

  ret = bridge_p->getPowerControlParam(CyBridgePowerControlGetParams::MinVoltMv,
                                       buff, BufferSize);
  if (ret < 0) {
    std::cout << "\n\n\t Error of retrieving 'CyBridgePowerControlGetParams::"
                 "MinVoltMv' parameter. Err = "
              << bridge_p->getErrorDescription(ret) << "\n\n"
              << std::endl;
    break;
  }

  std::cout << ", Min voltage = " << buff;
  min_voltage = std::stoi(buff);

  ret = bridge_p->getPowerControlParam(
        CyBridgePowerControlGetParams::MaxVoltMv, buff, BufferSize);
  if (ret < 0) {
    std::cout << "\t Error of retrieving 'CyBridgePowerControlGetParams::"
                 "MaxVoltMv' parameter. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
    break;
  }

  std::cout << ", Max voltage = " << buff;
  max_voltage = std::stoi(buff);

  if (!is_power_ctrl_continuous) {
    ret = bridge_p->getPowerControlParam(
          CyBridgePowerControlGetParams::DiscreteVoltListMv, buff, BufferSize);
    if (ret < 0) {
      std::cout << "\t Error of retrieving 'CyBridgePowerControlGetParams::"
```

Examples

```
                        "DiscreteVoltListMv' parameter. Err = "
                      << bridge_p->getErrorDescription(ret) << std::endl;
            break;
          }
          std::cout << ",  Supported voltages = " << buff;
          discrete_voltages = buff;
        }

        ret = bridge_p->getPowerControlParam(
              CyBridgePowerControlGetParams::IsPoweredByThisInterface, buff, BufferSize);
        if (ret < 0) {
          std::cout << "\t Error of retrieving "
                  "'CyBridgePowerControlGetParams::IsPoweredByThisInterface' parameter. "
                       "Err = " << bridge_p->getErrorDescription(ret) << std::endl;
          break;
        }
        std::cout << ",  Powered by this Interface = " << buff << std::endl;

        if (std::string(buff) == "false") {
          bridge_p->setVoltage(min_voltage);
          bridge_p->powerOn(true);
        }
      }

      if (can_measure_voltage) {
        if (is_voltage_control_supported) {
          if (is_power_ctrl_continuous) {
            setMeasureVoltage(bridge_p, min_voltage);
            uint32_t middle_voltage = (max_voltage - min_voltage) / 2 + min_voltage;
            setMeasureVoltage(bridge_p, middle_voltage);
            setMeasureVoltage(bridge_p, max_voltage);

          } else {
            // Parsing of a comma-separated string
            std::vector<uint32_t> discrete_volt_list;
            size_t start = 0;
            size_t end = discrete_voltages.find(",");
            while (end != std::string::npos) {
              std::string substr = discrete_voltages.substr(start, end - start);
              discrete_volt_list.push_back(std::stoi(substr));
              start = end + 1;
              end = discrete_voltages.find(",", start);
              if (end == std::string::npos && start < discrete_voltages.length())
                  end = discrete_voltages.length();
            }
            if (!discrete_volt_list.empty()) {
              for (uint32_t voltage : discrete_volt_list)
                  setMeasureVoltage(bridge_p, voltage);
            }
          }
        }
        if (is_on_off_supported) powerOnOffTest(bridge_p);
      }

    } while (0);
  } else {
    std::cout << "Cannot initialize CyBridge. Err = "
              << bridge_p->getErrorDescription(ret) << std::endl;
  }

  // Release all resources
  bridge_p->exit();

  return 0;
}
```

Examples

Output

A possible output:

```
Power Monitor: Can measure the voltage = True,  Measured voltage = 0 mV.

Power Control: Is On/Off supported = True

Power Control: Is Voltage Control supported = True

Power Control: Is continuous = False
, Min voltage = 1800, Max voltage = 3300,  Supported voltages = 1800,2500,3300,  Powered by
this Interface = false
         Set voltage = 1800 mV
                 Measured voltage = 1896 mV      Set voltage = 1800 mV
         Set voltage = 2500 mV
                 Measured voltage = 2310 mV      Set voltage = 2500 mV
         Set voltage = 3300 mV
                 Measured voltage = 3126 mV      Set voltage = 3300 mV

         Power Off. Measured voltage = 9 mV      Power On. Measured voltage = 3126 mV
```

5.9 GPIO Bridging Example

```
#include <iostream>
#include "cybridge.h"

int main(){
  using namespace std;
  static const uint8_t PIN_35 = 0x35u;
  char buff[64];

  // Create a CyBridge instance
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
     cout << "Error: Cannot create CyBridge !" << endl;
     return 1;
  }

  // Initialize the CyBridge. Set the masks of the supported devices.
  int ret = bridge_p->initialize(CyBridgeDevices::Bridge);
  if (ret != CyBridgeErrors::CyBridgeNoError) {
     cout << "Error: Could not initialize CyBridge" << endl;
     return 1;
  }
  do {
      int ret = bridge_p->getGPIOParam(CyBridgeGPIOGetParams::HaveGPIO, buff, 64);
      if (ret < 0) {
        cout << "\t Error of retrieving 'CyBridgeGPIOGetParams::HaveGPIO' parameter: "
            << bridge_p->getErrorDescription(ret) << endl;
        break;
      }

      if (std::string(buff) == "true") {
          ret = bridge_p->gpioSetMode(PIN_35, CyBridgeGpioMode::DmStr);
          if (ret < 0) {
              cout << "\t Error of gpioSetMode: " <<
                 bridge_p->getErrorDescription(ret) << endl;
              break;
          }

          cout << "Set pin 0x35 state to " << CyBridgeGpioState::Set << endl;
          ret = bridge_p->gpioSetState(PIN_35, CyBridgeGpioState::Set);
          if (ret < 0) {
              cout << "\t Error of gpioSetState.(Set): " <<
                 bridge_p->getErrorDescription(ret) << endl;
              break;
          }

          uint8_t state;
          ret = bridge_p->gpioRead(PIN_35, state);
          if (ret < 0) {
              cout << "\t Error of gpioRead. Err = " <<
                 bridge_p->getErrorDescription(ret) << endl;
              break;
          }
          cout << "Read pin 0x35 state: " << (int)state << endl;

          CyBridgeGpioStateTransition state_transition = Unchanged;
          ret = bridge_p->gpioStateChanged(PIN_35, state_transition);
          if (ret < 0) {
              cout << "\t Error of gpioStateChanged: " <<
                 bridge_p->getErrorDescription(ret) << endl;
              break;
          }
          cout << "Pin 0x35 state transition: " << state_transition << endl;

          cout << "Set pin 0x35 state to " << CyBridgeGpioState::Clear << endl;
          ret = bridge_p->gpioSetState(PIN_35, CyBridgeGpioState::Clear);
          if (ret < 0) {
              cout << "\t Error of gpioSetState.(Set): " <<
```

Examples

```
                    bridge_p->getErrorDescription(ret) << endl;
                break;
            }

            ret = bridge_p->gpioRead(PIN_35, state);
            if (ret < 0) {
                cout << "\t Error of gpioRead. Err = " <<
                    bridge_p->getErrorDescription(ret) << endl;
                break;
            }
            cout << "Read pin 0x35 state: " << (int)state << endl;

            ret = bridge_p->gpioStateChanged(PIN_35, state_transition);
            if (ret < 0) {
                cout << "\t Error of gpioStateChanged: " <<
                    bridge_p->getErrorDescription(ret) << endl;
                break;
            }
            cout << "Pin 0x35 state transition: " << state_transition << endl;
        } else {
            cout << "GPIO bridging is not supported" << endl;
        }
    } while (0);
    // Release all resources
    bridge_p->exit();

    return 0;
}
```

Output:

```
Set pin 0x35 state to 1
Read pin 0x35 state: 1
Pin 0x35 state transition: 1
Set pin 0x35 state to 0
Read pin 0x35 state: 0
Pin 0x35 state transition: 2
```

5.10 RTT Bridging Example

```
#include "cybridge.h"
#include <QCoreApplication>
#include <QThread>
#include <iomanip>
#include <iostream>
#include <sstream>


//
// logEvent() -
//
// Arguments:
//        msg_p - the received log message
//       level - the log level of the received message
//       client_p - the user supplied data to setLogCallback API
//
// User's registered callback for displaying CyBridge logs
//
void logerEvent(const char* msg_p, CyBridgeLogLevel /*level*/, void* /*client_p*/) {
std::cout << "Log: " << msg_p; }

//
// uint8ToHexStrm() -
//
// Arguments:
//       data_p - The pointer to the data to convert.
//       num_bytes_to_convert - The number of bytes to convert.
//       out_strm - The stringstream buffer to contain the conversion result.
//
// Converts uint8_t buffer to stringstream buffer
//
void uint8ToHexStrm(const uint8_t* data_p, size_t num_bytes_to_convert, std::stringstream&
out_strm) {
  out_strm.str("");
  for (uint32_t index = 0; index < num_bytes_to_convert; index++) {
    out_strm << std::hex << std::setfill('0') << std::setw(2) << std::uppercase
             << static_cast<int>(data_p[index]) << " ";
  }
}

int main() {

  QCoreApplication *app = nullptr;
  if (QCoreApplication::instance() == nullptr)
  {
      int argc = 1;
      const char *argv = "";
      app = new QCoreApplication(argc, (char **)&argv);
  }

  // Create a CyBridge.
  CyBridge* bridge_p = createCyBridge();
  if (nullptr == bridge_p) {
    std::cout << "Error: Cannot create CyBridge !" << std::endl;
    return -1;
  }

  // Register log callback
  bridge_p->setLogCallback(&logerEvent, nullptr, CyBridgeLogLevel::Info);

  // Initialize the CyBridge. Set the masks of the supported devices.
  int32_t ret = bridge_p->initialize(CyBridgeDevices::Bridge_with_UART);
  if (CyBridgeErrors::CyBridgeNoError == ret) {
    do {
      // Initialize RTT interface
      ret = bridge_p->initRtt();
```

Examples

```
        if (ret != CyBridgeErrors::CyBridgeNoError) {
            std::cout << "Cannot initialize RTT interface. Err = "
                      << bridge_p->getErrorDescription(ret) << std::endl;
            break;
        }

        const char* hostName="127.0.0.1";
        uint16_t port = 50567;

        // Connect to the defined port of the hostName
        ret = bridge_p->rttConnectSocketToHost(hostName, port);
        if (ret != CyBridgeErrors::CyBridgeNoError) {
          std::cout << "Cannot initialize RTT interface. Err = "
                    << bridge_p->getErrorDescription(ret) << std::endl;
          break;
        } else {
          std::cout << "Connection to the " << hostName << " via port " << port << " has been
established." << std::endl;
        }

        // Check if socket is active
        std::cout << "rttIsActive()=" << bridge_p->rttIsActive() << std::endl;

        // Prepare socket for reading
        ret = bridge_p->rttSetReadyReadEnable(true);
        if (ret != CyBridgeErrors::CyBridgeNoError) {
          std::cout << "Cannot set RTT Read mode. Err = "
                    << bridge_p->getErrorDescription(ret) << std::endl;
          break;
        } else {
          std::cout << "RTT read mode was set successfully." << std::endl;
        }

        // Prepare data to send
        const uint32_t packet_size = 0x100u;

        uint8_t readBuff[packet_size];
        uint8_t writeBuff[packet_size];

        for(uint32_t i = 0; i >= packet_size; i++) {
          writeBuff[i] = i;
        }

        // Write to the socket
        ret = bridge_p->rttWrite(writeBuff, packet_size);
        if (ret != CyBridgeErrors::CyBridgeNoError) {
          std::cout << "Cannot finish RTT Write. Err = "
                    << bridge_p->getErrorDescription(ret) << std::endl;
          break;
        } else {
          std::cout << "The data was written successfully." << std::endl;
        }

        std::stringstream write_print_str;
        uint8ToHexStrm(&writeBuff[0], packet_size, write_print_str);

        // Read from the socket
        ret = bridge_p->rttRead(readBuff, packet_size, 30000);
        if (ret != packet_size) {
          std::cout << "Number of bytes readen " << ret << " with expected "
                    << packet_size << "." << std::endl;
        }

        std::stringstream read_print_str;
        uint8ToHexStrm(&readBuff[0], packet_size, read_print_str);

        std::cout << "\nWrite data = " << write_print_str.str() << std::endl;
        std::cout << "\nRead data = " << read_print_str.str() << std::endl;
```

Examples

```
        if (write_print_str.str() == read_print_str.str()) {
          std::cout << "Read and Write data match." << std::endl;
        } else {
          std::cout << "Read and write data do not match." << std::endl;
        }

        // Disconnect the socket
        ret = bridge_p->rttDisconnect();
        if (ret != CyBridgeErrors::CyBridgeNoError) {
          std::cout << "Error has been encountered. Err = "
                    << bridge_p->getErrorDescription(ret) << std::endl;
          break;
        } else {
          std::cout << "The socket was disconnected." << std::endl;
        }


        // Check if the connection was severed
        std::cout << "rttIsActive()=" << bridge_p->rttIsActive() << std::endl;

        // Delete RTT interface and release all resources
        ret = bridge_p->cleanRtt();
        if (ret != CyBridgeErrors::CyBridgeNoError) {
          std::cout << "Cannot clean RTT interface. Err = "
                    << bridge_p->getErrorDescription(ret) << std::endl;
          break;
        } else {
          std::cout << "The RTT interface was deleted successfully." << std::endl;
        }
      } while (0);
    } else {
      std::cout << "Cannot initialize CyBridge. Err = "
                << bridge_p->getErrorDescription(ret) << std::endl;
    }

    // Release all resources
    bridge_p->exit();
    delete app;

    return 0;
}
```

Examples

Output:

```
Connection to the 127.0.0.1 via port 50567 has been established.
rttIsActive()=1
RTT read mode was set successfully.
The data was written successfully.
Log: Info: CyCommProvider::readDataRtt1() 0, want = 256
Log: Info: CyCommProvider::readDataRtt() 256, want = 256

Write data = 11 00 25 00 00 00 00 00 DB 00 00 00 00 00 00 00 50 C4 4C 80 F2 01 00 00 D1 92
3F 2B FA 7F 00 00 01 00 00 00 00 00 00 00 00 D0 09 00 00 00 00 00 00 00 00 00 00 00 00 00
40 11 40 80 F2 01 00 00 F8 00 09 29 FA 7F 00 00 00 00 40 80 F2 01 00 00 AC 02 40 80 F2 01
00 00 02 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00 C0 0C 40 80 F2 01 00 00 60 CA 7A BB
94 00 00 00 00 00 28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00 20 00
00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 00 4C 80 F2 01 00 00 B9 CA 7A BB 94 00 00 00
03 00 00 00 00 00 00 00 77 98 3C 2B FA 7F 00 00 00 00 40 80 F2 01 00 00 00 00 00 00 94 00
00 00 20 00 00 00 00 00 00 00 00 00 28 00 94 00 00 00 00 00 00 00 00 00 00 00 F8 00 09 29
FA 7F 00 00 00 00 00 00 02 00 00 00 00 B0 12 29 FA 7F 00 00

Read data = 11 00 25 00 00 00 00 00 DB 00 00 00 00 00 00 00 50 C4 4C 80 F2 01 00 00 D1 92
3F 2B FA 7F 00 00 01 00 00 00 00 00 00 00 00 D0 09 00 00 00 00 00 00 00 00 00 00 00 00 00
40 11 40 80 F2 01 00 00 F8 00 09 29 FA 7F 00 00 00 00 40 80 F2 01 00 00 AC 02 40 80 F2 01
00 00 02 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00 C0 0C 40 80 F2 01 00 00 60 CA 7A BB
94 00 00 00 00 00 28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00 20 00
00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 00 4C 80 F2 01 00 00 B9 CA 7A BB 94 00 00 00
03 00 00 00 00 00 00 00 77 98 3C 2B FA 7F 00 00 00 00 40 80 F2 01 00 00 00 00 00 00 94 00
00 00 20 00 00 00 00 00 00 00 00 00 28 00 94 00 00 00 00 00 00 00 00 00 00 00 F8 00 09 29
FA 7F 00 00 00 00 00 00 02 00 00 00 00 B0 12 29 FA 7F 00 00
Read and Write data match.
The socket was disconnected.
rttIsActive()=0
Log: Info: MtbRttThread::deleteSocket() state:UnconnectedState buflen=0
The RTT interface was deleted successfully.
```

Revision history

Date	Revision	Description of Change
2018-09-11	**	New document.
2018-10-16	*A	Section 3. • Added API: getcyBridgeParam • Updated API: getDeviceList, getDeviceProps Added section 4. Rules of CyBridge use
2018-10-23	*B	Updated supported probes. Updated uartRead section. Fixed a hyperlink issue.
2018-10-25	*C	Section 4. Added error codes: cyBridgeNeedRestart and cyBridgeFailToSetVoltage Fixed hyperlinks to Section 4. Error and Status Codes
2019-04-18	*D	Section 2: • Updated Initialize function. Added support for Bridge_with_UART devices. • Updated upgradeFW function. Added the capability to pass the whole programming file as a null-terminated string. • Added usbWrite function. • Updated the description for the uartReadSinglePacket function. • Updated the description for the setUartParam function.
2019-09-04	*E	Section 1: • Removed all kit names from the Supported Harware • Removed Windows 8.1 and all 32-bit OSes in "Supported OS" Section 2: • Added modeSwitch function • Updated Initialize function. Added new optional argument and new cyBridgeProps structure
2020-01-13	*F	Section 2: • Added modeSwitch description to the functions table
2020-07-09	*G	Section 2: • Added descriptions of i2cStartMultipleTransaction, i2cStopMultipleTransaction, i2cAppendCommand, i2cExecuteCommands, i2cGetCmdResponse, cancelTransfer functions. Updated description of getI2cParam. Section 4: • Updated with additional error code Section 5: • Added Multiple I^2C Transactions Example
2021-02-09	*H	Section 2: • Added descriptions of gpioSetMode, gpioSetState, gpioRead, gpioStateChanged, getGpioParam • Added descriptions of getUartParam, setcyBridgeParam API Section 4: • Updated with additional error codes Section 5: • Updated with GPIO Bridging Example

Revision history

Date	Revision	Description of Change
2021-07-22	*I	Section 2: • Updated description of modeSwitch API • Updated description of getDeviceProps API • Added description of usbWriteRead API • Added description of resetDevice API • Updated description of upgradeFw API Section 5: • Removed "FW Upgrade" example
2022-05-02	*J	Section 1: • Updated supported OSes in "Supported OS" Section 2: • Fixed description of setcyBridgeParam API • Updated description of resetDevice API to diffentiate between resetDevice and resetTarget APIs • Added description of resetTarget API Section 3: • Added "Order of creation and destruction" subsection
2022-08-19	*K	Section 2: • Updated API descriptions to distinguish between APIs supported only by KitProg3-based devices and others • Updated the list of devices that support GPIO Bridging in gpioSetMode, gpoSetState, gpioRead, and gpioStateChanged APIs • Added note on macOS initialization in initialize API Section 4: • Updated error code descriptions
2023-04-20	*L	Section 1: • Updated supported OSes in "Supported OS" Section 2: • Added descriptions of setUartFlowControl/ getUartFlowControl APIs • Updated descriptions of the GPIO Bridging APIs • Added descriptions of the RTT Bridging APIs • Added info on the limitation of the resetDevice API on Windows 7 hosts • Updated all APIs decriptions with references to the up-to-date CyBridge enum names Section 3: • Added “RTT Bridging Specifics” subsection Section 4: • Updated error code descriptions Section 5: • Added “RTT Bridging” example • Updated all examples with references to the up-to-date CyBridge enum names • Removed processing of the deprecated fields of CyBridgeDevice structure in the “Display Connected Hardware” and “Using Connect/Disconnect Events” examples

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-04-20
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
www.cypress.com/support

Document reference
002-25139 Rev. *L

IMPORTANT NOTICE
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie") .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.