lib_device_control: Device Control for xcore

Publication Date: 2026/3/26
Document Number: XM-010892-UG v5.0.1

XMOS

IN THIS DOCUMENT

1 Introduction

The Device Control Library is a protocol layer that handles the routing of control messages between a host and one or many controllable resources within the controlled device as shown in Fig. 1. The library is transport agnostic and can be used with physical transports such as I2C, SPI, USB or XSCOPE.

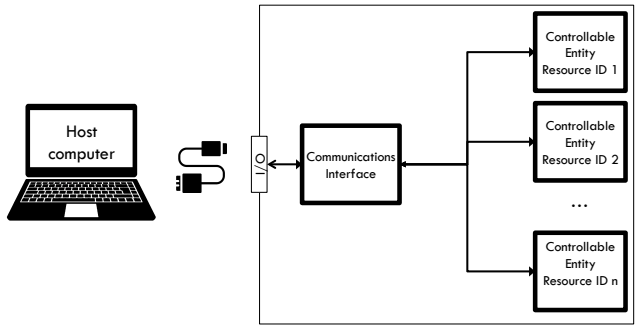

Fig. 1: Logical view of lib_device_control

All communications are fully acknowledged and so the host will be informed whether or not the device has correctly received or provided the required control information.

See `README` for currently supported combinations of host and transport mechanisms.

2 Usage

`lib_device_control` is intended to be used with the XCommon CMake, the *XMOS* application build and dependency management system.

To use this library in an application include `lib_device_control` in the application's `APP_DEPENDENT_MODULES` list in *CMakeLists.txt*, for example:

```
set(APP_DEPENDENT_MODULES "lib_device_control")
```

> **Note**
>
> Dependent modules should be pinned to release versions where possible, otherwise the latest commit on the *develop* branch will be used. For further details on managing modules, pinning to a release version and other options, please see the page xcommon-cmake Dependency Management.

All `lib_device_control` device functions can be accessed via the `control.h` header file, for example:

```
#include "control.h"
```

To select the transport layer for the device application, the preferred option is to create a control config header file `control_conf.h` in the application source directory and define the appropriate transport macro, for example:

```
#define CONTROL_USE_USB 1
```

> **Note**
>
> Please ensure the path to `control_conf.h` is included in the application's include path list, `APP_INCLUDES`.

Alternatively, the transport macro can be defined in the application's `CMakeLists.txt` file using the `APP_COMPILER_FLAGS`, for example:

```
set(APP_COMPILER_FLAGS ... -DCONTROL_USE_USB=1 ...)
```

2.1 Host Usage

The build system used for the host is native CMake, so the process for using the library on the host is slightly different to the device side.

Instead of setting the `APP_DEPENDENT_MODULES` variable, the host application must include the appropriate `CMake` file for the intended transport. The transport-specific `CMake` file will set up the necessary source and include paths and link against the appropriate transport library for the host application. The files are found in `lib_device_control/host`, for example `host_build_usb.cmake` for USB transport.

```
include("${CMAKE_CURRENT_LIST_DIR}/../../../host/host_build_usb.cmake")
```

The host application should then link against the provided transport library, for example `control_usb_host` for USB transport, after calling `add_executable()` for the host application target in *CMakeLists.txt*, for example:

```
add_executable(usb_host_app "src/host.c")

target_link_libraries(usb_host_app PRIVATE control_usb_host)
```

The `lib_device_control` host functions can be accessed via the `control_host.h` header file, for example:

```
#include "control_host.h"
```

2.1.1 Host dependencies

For Windows hosts, the supported compiler is *MSVC*. The Ninja build system is recommended to be used with CMake, but it is not required. The `libusb` library is available via the `host_build_usb.cmake` file.

For Linux hosts, including Raspberry Pi, the supported compiler is *GCC*. The `libusb-1.0-0-dev` package must be installed for USB transport, and the the `host_build_usb.cmake` file will link against the library.

For OSX hosts, the supported compiler is *Clang*. The `libusb` library is available via the `host_build_usb.cmake` file.

3 Operation

The *Host* communicates with resources on an XCORE device by sending *commands* to it over a physical *transport*, as shown in Fig. 2. Resources are identified by an 8-bit identifier and exist in tasks that run on threads of the device. There can be multiple resources in a task.

The command code is 8 bits and is a *write* command when bit 7 is not set or a *read* command when bit 7 is set. It is an application design decision whether the commands are common across all resources or unique to each resource, but the library provides a simple mechanism for routing commands to the correct resource based on the resource ID.

The length field is 8 bits and indicates the number of bytes of data related to the command, which can be zero.

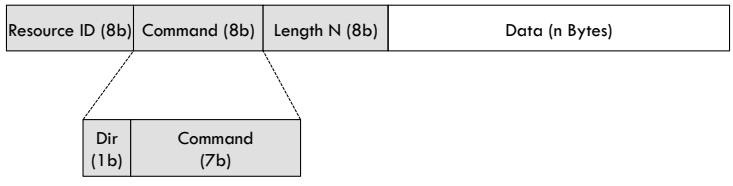

Fig. 2: Packet for control communications

Read and write Commands can include *data* bytes that are optional, but the number of data bytes transferred must equal the length field.

When starting a new request, the *resource ID*, *command code* and *length* fields are all sent by the host for read and write commands, but the *data* bytes are only sent by the host for write commands. For read commands, the device sends the *data* bytes back to the host.

There is a transport task in the device (e.g. I2C slave or USB endpoint 0) that dispatches all commands. All other tasks that have resources connect to this transport task over the `control` interface.

Tasks *register* their resources during initialisation when requested over the interface.

When commands are received by the transport task they forwarded over the matching `control` interface. The mapping of resources to tasks is done by the application and the library simply forwards the command to the appropriate task based on the resource ID, as shown in the figure Fig. 3:

This means multiple tasks residing in different threads or even tiles on the device can be easily controlled using a single instance of the Device Control library and a single API from the host.

Commands have a result code to indicate success or failure. The result is propagated to the host so the host can indicate whether an error occurred to the user.

The control library supports USB (device is USB device), I2C (device is I2C slave), SPI (device is SPI slave) and XSCOPE (device is target connected via XTAG debug adapter) as physical layers. The maximum data packet size for each of the transport types is as shown in Table 1.

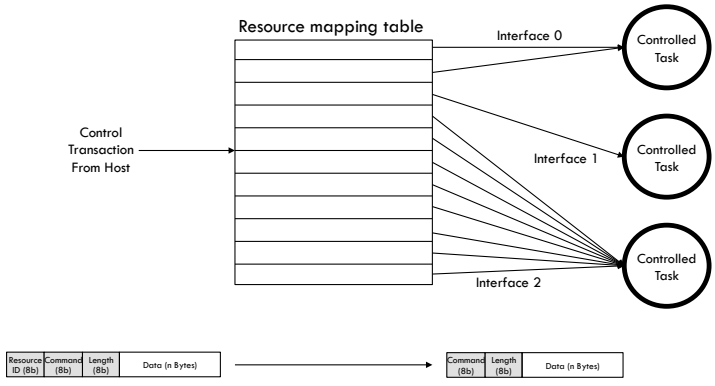

Fig. 3: Mapping between resource IDs and XC interfaces

Table 1: Maximum Data Length for Device Control Library Transports

Transport	Data length	Limitation
I2C	253 Bytes	Arbitrary
USB	64 Bytes	USB control transfer specification
XSCOPE	252 Bytes	Arbitrary
SPI	253 Bytes	Arbitrary

It would be straightforward to add support for additional physical layers such as UART or TCP/UDP over Ethernet or add additional control hosts where the hardware and operating system supports it.

4 Transport

The transport task receives its natural unit of data, such as I2C transaction, or USB request, and calls a processing function on it from the library. At the same time it passes in the whole array of XC interfaces which connect to all of the controlled tasks.

The library's logic happens inside the function that is called and once a command is complete, an XC interface call is made to pass the command over to the appropriate controlled resource.

The receiving task then receives a write or read command over the XC interface.

To ensure compatibility, a special command is provided to query the version of control XC interface. This allows the host to query the device and check that it is running the same version, which will ensure command compatibility.

Please see *Device side Transport*, *Device side* and *Host side* sections for further details.

4.1 I2C, SPI and XSCOPE Transports

I2C, SPI and XSCOPE Transports are very similar in concept, so the I2C is given here as an example.

4.1.1 Allocating hardware resources

Allocating the necessary pins for the transport is the responsibility of the application. For example, for an I2C slave application, two ports must be allocated for SCL and SDA lines and passed to the I2C slave function in the transport task, for example:

```
on tile[PORT_I2C_SCL_TILE_NUM]: port p_scl = PORT_I2C_SCL;
on tile[PORT_I2C_SDA_TILE_NUM]: port p_sda = PORT_I2C_SDA;
```

4.1.2 Declaring interfaces

The application must declare an interface for the transport and for the control library, for example:

```
int main(void)
{
  i2c_slave_callback_if i_i2c;
  interface control i_control[CONTROL_INTERFACES_NUM];
```

4.1.3 Registering Controllable Resources

Before any control can take place the controlled tasks must register controllable resources. This is done by calling `control_register_resources()` at startup, before calling into the tasks.

```
      control_init();
      control_register_resources(i_control, CONTROL_INTERFACES_NUM);
```

The application side of the registration process is handled by populating a table of resource IDs at startup which is done by `register_resources()`.

```
      case i_control.register_resources(control_resid_t resources[MAX_RESOURCES_PER_INTERFACE],
                                        unsigned &num_resources):
        resources[0] = RESOURCE_ID;
        num_resources = 1;
        break;
```


4.1.4 Transport Tasks

The transport task must be called in main and provided the control interface, the control client must be called, `i2c_control_client` in this example:

```
      par {
#pragma warning disable unusual-code // Suppress slice interface warning (no array size passed)
        i2c_control_client(i_i2c, i_control);
#pragma warning enable
        i2c_slave(i_i2c, p_scl, p_sda, DEVICE_I2C_ADDRESS);
      }
```

4.1.5 Application Task

The application task must be called in main and provided the control interface, for example:

```
    on tile[PORT_I2C_SCL_TILE_NUM]: par {
      app(i_control[0]);
    }
```

4.1.6 Reading over Device Control

When the host requests a read from a controlled resource, the `read_command()` is called on the interface. The example is very simple, it checks the resource ID is correct then returns the last command that was written to the resource.

```
        if (resid != RESOURCE_ID) {
          printf("unrecognised resource ID %d\n", resid);
          ret = CONTROL_ERROR;
          break;
        }
        // Simple test: fill the payload with the last written value
        for (int i = 0; i < payload_len; i++) {
          payload[i] = test_value;
        }
        ret = CONTROL_SUCCESS;
        break;
```

4.1.7 Writing over Device Control

When the host requests a write to a controlled resource, the `write_command()` is called on the interface. The example is very simple, it checks the resource ID is correct then records the data on the resource.

```
        if (resid != RESOURCE_ID) {
          printf("unrecognised resource ID %d\n", resid);
          ret = CONTROL_ERROR;
          break;
        }
        if (payload_len > 0) {
          test_value = payload[0];
        } else {
          test_value = 0;
        }
        ret = CONTROL_SUCCESS;
        break;
```

4.2 USB Transport

The USB transport is slightly different to the I2C, SPI and XSCOPE transports as it is based on handling USB requests on endpoint 0. However, the same principles apply in terms of registering resources and handling read and write commands.

4.2.1 USB Application main() function

The `main()` function sets up the pins and tasks within the application. The difference to other transports is that there is typically an Endpoint0 task.

```
XUD_EpType epTypeTableOut[XUD_EP_COUNT_OUT] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE};
XUD_EpType epTypeTableIn[XUD_EP_COUNT_IN] =   {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE};

int main(void)
{
  chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];
  interface control i_control[1];
  par {
    on USB_TILE: par {
      Endpoint0(c_ep_out[0], c_ep_in[0], i_control);
      XUD_Main(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
                       null, epTypeTableOut, epTypeTableIn,
                       XUD_SPEED_HS, XUD_PWR_BUS);
    }
    on tile[0]: par {
      app(i_control[0]);
    }
  }
  return 0;
}
```

4.2.2 USB Descriptors

USB requires descriptors to describe the devices capabilities. The example provides only a single interface with only the control endpoint (0) and so the device descriptor and configuration descriptor are very simple. The device descriptor indicates that the device is a Vendor Class device, which means it will use Vendor requests for communication between the host and device, which are handled by the Device Control library. The USB device descriptor:

```
/* Device Descriptor */
static unsigned char devDesc[] =
```

And configuration descriptor:

```
/* Configuration Descriptor */
static unsigned char cfgDesc[] =
```

The descriptors are requested by the host during enumeration and are handled in the Endpoint0 task:

```
            if(result == XUD_RES_ERR)
            {
                result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
                             sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
                             NULL, 0,
                             NULL, 0,
                             stringDescriptors, sizeof(stringDescriptors)/sizeof(stringDescriptors[0]),
                             sp, usbBusSpeed);
            }
```

4.2.3 Reading over Device Control

When the host requests a read from a controlled resource, endpoint 0 receives a device-to-host Vendor request.

```
                case USB_BMREQ_D2H_VENDOR_DEV:
                    if (sp.bRequest == XUD_REQUEST_GET_MSOS_DESCRIPTOR) {
                        result = XUD_GetMsosDescriptor(ep0_out, ep0_in, &sp);

                    } else {
#pragma warning disable unusual-code // Suppress slice interface warning (no array size passed)
                        result = USB_D2H_VendorRequest(ep0_out, ep0_in, &sp, i_control);
#pragma warning enable
                    }
                    break;
```

The Device Control transport function `USB_D2H_VendorRequest()` takes the USB setup packet and channels along with the control interface and wraps the USB data transaction that is translated to a control command. A call to the `read_command()` method is subsequently made on the server side control interface which is then handled by the application. Once the application has filled the buffer with data, the data buffer reference is returned to `lib_xud` by EP0 and the USB control transaction is completed. The Vendor request transaction is stalled to indicate failure to the host.

```
        if (resid != RESOURCE_ID) {
          printf("unrecognised resource ID %d\n", resid);
          ret = CONTROL_ERROR;
          break;
        }
        // Simple test: fill the payload with the last written value
        for (int i = 0; i < payload_len; i++) {
          payload[i] = test_value;
        }
        ret = CONTROL_SUCCESS;
        break;
```

4.2.4 Writing over Device Control

When the host requests a write to a controlled resource, endpoint 0 receives a host-to-device Vendor request.

```
                case USB_BMREQ_H2D_VENDOR_DEV:
#pragma warning disable unusual-code // Suppress slice interface warning (no array size passed)
                    result = USB_H2D_VendorRequest(ep0_out, ep0_in, &sp, i_control);
#pragma warning enable
                    break;
```

The Device Control transport function `USB_H2D_VendorRequest()` takes the USB setup packet and channels along with the control interface and wraps the USB data transaction that is translated to a control command. The call to the `write_command()` method is subsequently made on the control interface which is then handled by server side in the application. The USB transaction is then either be acknowledged or stalled by the EP0 handler to indicate success or failure to the host.

```
        if (resid != RESOURCE_ID) {
          printf("unrecognised resource ID %d\n", resid);
          ret = CONTROL_ERROR;
          break;
        }
        if (payload_len > 0) {
          test_value = payload[0];
        } else {
          test_value = 0;
        }
        ret = CONTROL_SUCCESS;
        break;
```

4.2.5 Windows USB BOS/MSOS Support

Windows requires a special USB descriptor, the BOS/MSOS descriptor, to be able to use WinUSB as the driver for the device. This descriptor is provided by `lib_xud` and available by calling `XUD_GetBosDescriptor()` and `XUD_GetMsosDescriptor()`. The BOS descriptor handling is a standard device request:

```
                case USB_BMREQ_D2H_STANDARD_DEV:
                    if (sp.bRequest == USB_GET_DESCRIPTOR) {
                        result = XUD_GetBosDescriptor(ep0_out, ep0_in, &sp);
                    }
                    break;
```

The MSOS descriptor is requested by the host as a Vendor request:

```
                case USB_BMREQ_D2H_VENDOR_DEV:
                    if (sp.bRequest == XUD_REQUEST_GET_MSOS_DESCRIPTOR) {
                        result = XUD_GetMsosDescriptor(ep0_out, ep0_in, &sp);

                    } else {
```


5 Example application

5.1 Building the example

There are many example applications provided in the `examples` directory of the software package, which demonstrate how to use the library with different physical layers and on different host platforms. The following instructions are for the USB example, but the process is very similar for the I2C, SPI and XSCOPE examples.

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `lib_device_control` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd lib_device_control/examples/usb/device
cmake -G "Unix Makefiles" -B build
```

If any dependencies are missing they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `examples/usb/device/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

5.2 Running the example

From an XTC command prompt, the following command should be run from the `examples/usb/device` directory:

```
xrun --xscope ./bin/usb.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash ./bin/usb.xe
```

5.3 Building the example host app

This is very similar to building the device example, except the host example is in the `examples/usb/host` directory, and the host compiler must be in the path. The host app can be built from a command terminal with the commands shown in Listing 1.

Listing 1: Building the host app on Linux or Mac hosts

```
cd lib_device_control/examples/usb/host
cmake -G "Unix Makefiles" -B build
xmake -j -C build
./bin/usb_host_app
```

For Windows hosts the process is the same except the Ninja generator is recommended to be used with CMake and the executable will have a `.exe` extension. The commands are shown in Listing 2.

Listing 2: Building the host app on Windows hosts

```
cd lib_device_control\examples\usb\host
cmake -G "Ninja" -B build
cmake --build build
bin\usb_host_app.exe
```

5.4 Example Hardware Setup

To run the example, connect a USB cable to power the *XK-VOICE-L71* board as shown in Fig. 4, and plug the XTAG to the board and connect the XTAG USB cable to your development machine.

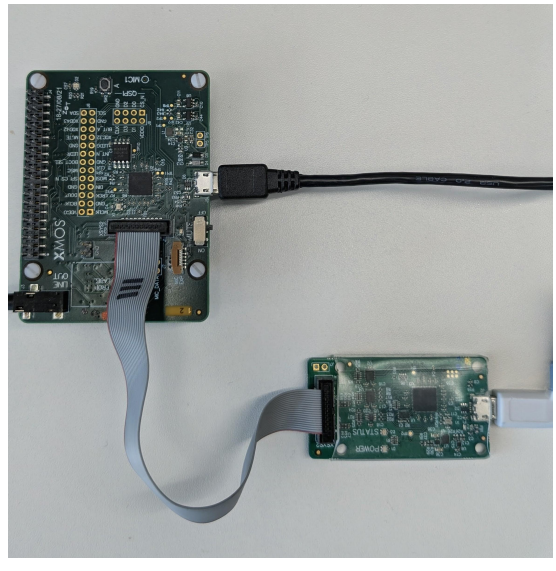

Fig. 4: XK-VOICE-L71 USB Hardware setup

5.4.1 Raspberry Pi Hardware Setup

For the examples with the Raspberry Pi as a host, the I2C, SPI and USB examples, the *XK-VOICE-L71* board can be stacked on top of a Raspberry Pi as shown in Fig. 5, and the I2C and SPI lines are connected between the two boards. For the USB example a USB cable should be connected between the Raspberry Pi and the *XK-VOICE-L71*. The XTAG should be connected to the *XK-VOICE-L71* as normal and the host app can be run from the Raspberry Pi terminal.

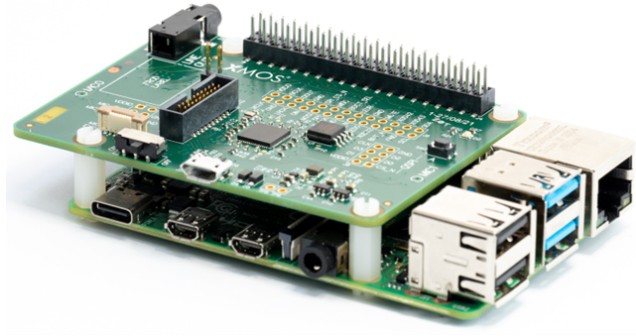

Fig. 5: XK-VOICE-L71 Raspberry Pi Hardware setup

5.4.2 Output from the example applications

When run, the host app will attempt to connect and communicate with the device over USB. The outputs listings are shown in Listing 3 and Listing 4 for the device and host applications respectively.

Listing 3: Example device app output

```
> xrun --xscope .\bin\usb.xe
started
1: W 18 0 60, 00 02 03 04 05 06 07 08 09 ... ``PAYLOAD_SIZE`` bytes in total
2: R 18 128 60
3: W 18 0 60, 01 02 03 04 05 06 07 08 09 ... ``PAYLOAD_SIZE`` bytes in total
4: R 18 128 60
5: W 18 0 60, 02 02 03 04 05 06 07 08 09 ... ``PAYLOAD_SIZE`` bytes in total
6: R 18 128 60
7: W 18 0 60, 03 02 03 04 05 06 07 08 09 ... ``PAYLOAD_SIZE`` bytes in total
8: R 18 128 60
```

Listing 4: Example host app output

```
$ sudo ./bin/usb_host_app
device found
started
Written and read back command with payload: 0x00
Written and read back command with payload: 0x01
Written and read back command with payload: 0x02
Written and read back command with payload: 0x03
done
```

The number of bytes sent by the host can be changed with the `PAYLOAD_SIZE` macro in the host app's `host.c` file.

```
#define PAYLOAD_SIZE 60
```

5.5 Example XSCOPE Application

The process of running the XSCOPE example differs from the other examples due to the data passing over the XSCOPE transport and accessed via a local Ethernet XSCOPE server.

Note

In order to build the XSCOPE host app the on the host machine, both the hosts' native toolchain and the XTC tools must be installed and in the path, then the host app should be built as described in the previous section.

Once the device is built, from an XTC command prompt, the following command should be run from the `examples/xscope/device` directory, please note the use of the `--xscope-port` option to specify the port for the local XSCOPE server:

```
xrun --xscope-port localhost:10101 .\bin\xscope.xe
```

When run, the host app will attempt to connect and communicate with the device over XSCOPE. As shown in Listing 5 and Listing 6, the device prints out very little, but the host app prints out the commands sent and responses received over XSCOPE.

Listing 5: Example XSCOPE device app output

```
> xrun --xscope-port localhost:10101 .\bin\xscope.xe
New (non-blocking) xscope_server_socket connection made
```

Listing 6: Example XSCOPE host app output

```
>bin\xscope_host_app.exe
[HOST] connected to server at port 10101
[HOST] device found
[HOST] 0 send version command: 00 80 01 72
started
XSCOPE server connected
[HOST] version response: 00 80 01 00 10
[HOST] started
[HOST] 1 write: 12 00 01 72 00
1: W 18 0 1, 00
[HOST] response: 12 00 01 00
[HOST] 2 read, len 4: 12 80 01 72
2: R 18 128 1
[HOST] read response: length 5: 12 80 01 00 00
[HOST] Written and read back command with payload: 0x00
...
[HOST] 7 write: 12 00 01 72 03
7: W 18 0 1, 03
[HOST] response: 12 00 01 00
[HOST] 8 read, len 4: 12 80 01 72
8: R 18 128 1
[HOST] read response: length 5: 12 80 01 00 03
[HOST] Written and read back command with payload: 0x03
[HOST] done
```


6 References

6.1 I2C

- lib_i2c (https://www.xmos.com/libraries/lib_i2c)
- https://developer.mbed.org/users/okano/notebook/i2c-access-examples
- http://www.robot-electronics.co.uk/i2c-tutorial
- https://www.raspberrypi.org/forums/viewtopic.php?f=44&t=15840&start=25

6.2 SPI

- lib_spi (https://www.xmos.com/libraries/lib_spi)

6.3 USB

- lib_xud (https://www.xmos.com/libraries/lib_xud)
- http://www.beyondlogic.org/usbnutshell/usb6.shtml

6.4 XSCOPE

- XSCOPE (https://www.xmos.com/documentation/XM-014363-PC/html/tools-guide/tools-ref/xscope/index.html#xscope)

7 API Reference

7.1 Shared

group **control_shared**

The shared API for using the device control library on the device and host

Defines

CONTROL_VERSION

This is the version of control protocol. Used to check compatibility

CONTROL_VENDOR_REQUEST

Expected USB Vendor request value on the control interface

Typedefs

typedef uint8_t **control_resid_t**

Type used to identify a resource

typedef uint8_t **control_cmd_t**

Type used to identify a command

typedef uint8_t **control_version_t**

Type used to identify a version

typedef uint8_t **control_ret_t**

Type used to identify a return value

typedef uint8_t **control_status_t**

Type used to identify a status

Enums

enum **control_ret_values**

This type enumerates the possible outcomes from a control transaction.
Values:

enumerator **CONTROL_SUCCESS**

enumerator **CONTROL_REGISTRATION_FAILED**

enumerator **CONTROL_BAD_COMMAND**

enumerator **CONTROL_DATA_LENGTH_ERROR**

enumerator **CONTROL_OTHER_TRANSPORT_ERROR**

enumerator **CONTROL_ERROR**

group **control_transport_shared**

The internal defines and structs for using the device control library on the device and host

Defines

IS_CONTROL_CMD_READ(c)

Checks if the read bit is set in a command code.

Parameters

- **c** – **[in]** The command code to check

Return values

- **true** – if the read bit in the command is set
- **false** – if the read bit is not set

CONTROL_CMD_VALUE(c)

Returns the application command value with the read/write bit cleared.

Parameters

- **c** – The transport command code.

Returns

The application command code, the read/write bit is cleared.

CONTROL_CMD_SET_READ(c)

Sets the read bit on a command code

Parameters

- **c** – The application command code to set the read bit on.

Returns

The transport command code, the read/write bit is set.

CONTROL_CMD_SET_WRITE(c)

Clears the read bit on a command code

Parameters

- **c** – The application command code to clear the read bit on.

Returns

The transport command code, the read/write bit is cleared.

CONTROL_SPECIAL_RESID

This is the special resource ID owned by the control library. It can be used to check the version of the control protocol. Services may not register this resource ID.

CONTROL_GET_VERSION

The command to read the version of the control protocol. It must be sent to resource ID CONTROL_SPECIAL_RESID.

CONTROL_GET_LAST_COMMAND_STATUS

The command to read the return status of the last command. It must be sent to resource ID CONTROL_SPECIAL_RESID.

7.2 Device side Configuration

group **Control_conf**

Definitions for the control configuration.

Defines

CONTROL_INTERFACES_NUM

Number of control server-client interfaces supported by the control library.

CONTROL_USE_USB

Enable USB transport. Add calls to USB_H2D_VendorRequest and USB_D2H_VendorRequest in the applications' `Endpoint0` thread

CONTROL_USE_I2C

Enable I2C transport. Add a call to i2c_control_client in `main()`

CONTROL_USE_SPI

Enable SPI transport. Add a call to spi_control_client in `main()`

CONTROL_USE_XSCOPE

Enable xSCOPE transport. Add a call to xscope_control_client in `main()`

RESOURCE_ID_DFU

Resource ID for DFU

MAX_RESOURCES_PER_INTERFACE

Resource count limits. Sets the size of the arrays used for storing the mappings

7.3 Device side

group **Control**

This interface is used to communicate with the control library from the application

Functions

void **register_resources**(
control_resid_t resources[*MAX_RESOURCES_PER_INTERFACE*], REFERENCE_PARAM(unsigned, num_resources),
)

Request from host to register controllable resources with the control library. This is called once at startup and is necessary before control can take place.

Parameters

- **resources** – Array of resource IDs of size MAX_RESOURCES_PER_INTERFACE
- **num_resources** – Number of resources populated within the resources[] table

control_ret_t **write_command**(
control_resid_t resid, *control_cmd_t* cmd, const uint8_t payload[*payload_len*], unsigned payload_len,
)

Request from host to write to controllable resource in the device. The command consists of a resource ID, command and a byte payload of length payload_len.

Parameters

- **resid** – Resource ID. Indicates which resource the command is intended for
- **cmd** – Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indicates a write command
- **payload** – Array of bytes which constitutes the data payload
- **payload_len** – Size of the payload in bytes

Returns

Whether the handling of the write data by the device was successful or not

control_ret_t **read_command**(*control_resid_t* resid, *control_cmd_t* cmd, uint8_t payload[*payload_len*], unsigned payload_len,
)

Request from host to read a controllable resource in the device. The command consists of a resource ID, command and a byte payload of length payload_len.

Parameters

- **resid** – Resource ID. Indicates which resource the command is intended for
- **cmd** – Command code. Note that this will be in the range 0x00 to 0x7F because bit 7 cleared indicates a read command
- **payload** – Array of bytes which constitutes the data payload
- **payload_len** – Size of the payload in bytes

Returns

Whether the handling of the read data by the device was successful or not

control_ret_t **control_init**(void)

Initialize the control library. Clears resource table to ensure nothing is registered.

Returns

Whether the initialization was successful or not

control_ret_t **control_register_resources**(
CLIENT_INTERFACE_ARRAY(control, i, *n*), unsigned n,
)

Sends a request to the application to register controllable resources.

Parameters

- **i** – Array of interfaces used to communicate with controllable entities
- **n** – The number of interfaces used

Returns

Whether the registration was successful or not

control_ret_t **control_process_i2c_write_start**(CLIENT_INTERFACE(control, i[]))

Inform the control library that an I2C slave write has started. Called from I2C callback API.

Parameters

- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the write start was successful or not

control_ret_t **control_process_i2c_read_start**(CLIENT_INTERFACE(control, i[]))

Inform the control library that an I2C slave read has started. Called from I2C callback API.

Parameters

- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the read start was successful or not

control_ret_t **control_process_i2c_write_data**(
const uint8_t data, CLIENT_INTERFACE(control, i[]),
)

Inform the control library that an I2C slave write has occurred. Called from I2C callback API.

Parameters

- **data** – Array of byte data to be passed to the device
- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the write was successful or not

control_ret_t **control_process_i2c_read_data**(
REFERENCE_PARAM(uint8_t, data), CLIENT_INTERFACE(control, i[]),
)

Inform the control library that an I2C slave read has occurred. Called from I2C callback API.

Parameters

- **data** – Reference to array of byte data to be passed back from the device
- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the read was successful or not

control_ret_t **control_process_i2c_stop**(CLIENT_INTERFACE(control, i[]))

Inform the control library that an I2C transaction has stopped. Called from I2C callback API.

Parameters

- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the stop was successful or not

control_ret_t **control_process_usb_set_request**(
uint16_t windex, uint16_t wvalue, uint16_t wlength, const uint8_t request_data[], CLIENT_INTERFACE(control, i[]),
)

Inform the control library that a USB set (write) has occurred. Called from USB EP0 handler.

Parameters

- **windex** – wIndex field from the USB Setup packet
- **wvalue** – wValue field from the USB Setup packet
- **wlength** – wLength field from the USB Setup packet
- **request_data** – Array of byte data to be written to the device
- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the write was successful or not

control_ret_t **control_process_usb_get_request**(
uint16_t windex, uint16_t wvalue, uint16_t wlength, uint8_t request_data[], CLIENT_INTERFACE(control, i[]),
)

Inform the control library that a USB get (read) has occurred. Called from USB EP0 handler.

Parameters

- **windex** – wIndex field from the USB Setup packet
- **wvalue** – wValue field from the USB Setup packet
- **wlength** – wLength field from the USB Setup packet
- **request_data** – Reference to array of byte data to be passed back from the device
- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the read was successful or not

control_ret_t **control_process_xscope_upload**(
uint8_t buf[], unsigned buf_size, unsigned length_in, REFERENCE_PARAM(unsigned, length_out), CLIENT_INTERFACE(control, i[]),
)

Inform the control library that an xscope transfer has occurred. Called from xscope handler. This function both reads and writes data in a single call. The data return is device (control library) initiated. Note: Data requires word alignment so we can cast to struct.

Parameters

- **buf** – Array of bytes for read and write data.
- **buf_size** – Array size in bytes
- **length_in** – Number of bytes to be written to device
- **length_out** – Number of bytes returned from device to be read by host
- **i** – Array of interfaces used to communicate with controllable entities

Returns

Whether the transfer was successful or not

7.4 Device side Transport

void **i2c_control_client**(
    SERVER_INTERFACE(i2c_slave_callback_if, i_i2c), CLIENT_INTERFACE(control, i_control[]),
)

I2C transport client processing function, this passes I2C data to the control interface.

This function handles I2C requests from the host. It should be called from **main()** and the interface linked to the I2C slave, in the case where CONTROL_USE_I2C is defined as 1.

Parameters

- **i_i2c** – I2C slave callback interface
- **i_control** – Control interface array

void **spi_control_client**(
    SERVER_INTERFACE(spi_slave_callback_if, i_spi), CLIENT_INTERFACE(control, i_control[]),
)

SPI transport client processing function, this passes SPI data to the control interface.

This function handles SPI requests from the host. It should be called from **main()** and the interface linked to the SPI slave, in the case where CONTROL_USE_SPI is defined as 1.

Parameters

- **i_spi** – SPI slave callback interface
- **i_control** – Control interface array

void **xscope_control_client**(
    chanend c_xscope, CLIENT_INTERFACE(control, i_control[]),
)

XSCOPE transport client processing function, this passes XSCOPE data to the control interface.

This function handles XSCOPE requests from the host. It should be called from **main()** and the interface linked to the XSCOPE channel, in the case where CONTROL_USE_XSCOPE is defined as 1.

Parameters

- **c_xscope** – XSCOPE channel end
- **i_control** – Control interface array

XUD_Result_t **USB_H2D_VendorRequest**(
    XUD_ep ep0_out, XUD_ep ep0_in, USB_SetupPacket_t *sp, CLIENT_INTERFACE(control, i_control[]),
)

USB transport host write (Set) processing function

This function handles USB vendor requests from the host to the device. It should be called from the Endpoint 0 thread of the application, in the case where CONTROL_USE_USB is defined as 1. The function receives the host data and passes it to the control interface for handling the request.

Parameters

- **ep0_out** – Endpoint 0 OUT endpoint
- **ep0_in** – Endpoint 0 IN endpoint
- **sp** – Pointer to USB setup packet
- **i_control** – Control interface array

Returns

XUD_Result_t Result of the operation

XUD_Result_t **USB_D2H_VendorRequest**(
XUD_ep ep0_out, XUD_ep ep0_in, USB_SetupPacket_t *sp, CLIENT_INTERFACE(control, i_control[]),
)

USB transport host read (Get) processing function

This function handles USB vendor requests from the device to the host. It should be called from the Endpoint 0 thread of the application, in the case where CONTROL_USE_USB is defined as 1. The function calls the control interface for handling the request, and sends data back to the host as needed.

Parameters

- **ep0_out** – Endpoint 0 OUT endpoint
- **ep0_in** – Endpoint 0 IN endpoint
- **sp** – Pointer to USB setup packet
- **i_control** – Control interface array

Returns

XUD_Result_t Result of the operation

7.5 Host side

control_ret_t **control_init_xscope**(const char *host_str, const char *port_str)

Initialize the xscope host interface

Parameters

- **host_str** – String containing the name of the xscope host. Eg. "localhost"
- **port_str** – String containing the port number of the xscope host

Returns

Whether the initialization was successful or not

control_ret_t **control_cleanup_xscope**(void)

Shutdown the xscope host interface

Returns

Whether the shutdown was successful or not

control_ret_t **control_init_i2c**(unsigned char i2c_slave_address)

Initialize the I2C host (master) interface

Parameters

- **i2c_slave_address** – I2C address of the slave (controlled device)

Returns

Whether the initialization was successful or not

control_ret_t **control_cleanup_i2c**(void)

Shutdown the I2C host (master) interface connection

Returns

Whether the shutdown was successful or not

control_ret_t **control_init_usb**(
int vendor_id, int product_id, int interface_num,
)

Initialize the USB host interface

Parameters

- **vendor_id** – Vendor ID of controlled USB device
- **product_id** – Product ID of controlled USB device
- **interface_num** – Deprecated parameter, no longer used

Returns

Whether the initialization was successful or not

control_ret_t **control_query_version**(*control_version_t* *version)

Checks to see that the version of control library in the device is the same as the host

Parameters

- **version** – Reference to control version variable that is set on this call

Returns

Whether the checking of control library version was successful or not

control_ret_t **control_write_command**(
control_resid_t resid, *control_cmd_t* cmd, const uint8_t payload[], size_t payload_len,
)

Request to write to controllable resource inside the device. The command consists of a resource ID, command and a byte payload of length payload_len.

Parameters

- **resid** – Resource ID. Indicates which resource the command is intended for
- **cmd** – Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indicates a write command
- **payload** – Array of bytes which constitutes the data payload
- **payload_len** – Size of the payload in bytes

Returns

Whether the write to the device was successful or not

control_ret_t **control_read_command**(
control_resid_t resid, *control_cmd_t* cmd, uint8_t payload[], size_t payload_len,
)

Request to read from controllable resource inside the device. The command consists of a resource ID, command and a byte payload of length payload_len.

Parameters

- **resid** – Resource ID. Indicates which resource the command is intended for
- **cmd** – Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indicates a write command
- **payload** – Array of bytes which constitutes the data payload
- **payload_len** – Size of the payload in bytes

Returns

Whether the read from the device was successful or not