lib_device_control: Device Control for xcore

Publication Date: 2026/3/12
Document Number: XM-010892-UG v5.0.0

XMOS

IN THIS DOCUMENT

1 Introduction

The Device Control Library handles the routing of control messages between a host and one or many controllable resources within the controlled device.

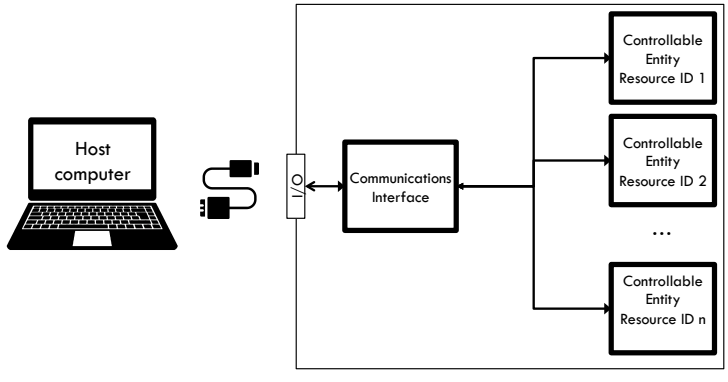

Fig. 1: Logical view of lib_device_control

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.

2 Operation

The *Host* controls resources on an XCORE device by sending *commands* to it over a *transport* protocol. Resources are identified by an 8-bit identifier and exist in tasks that run on logical cores of the device. There can be multiple resources in a task.

```
Send command c to resource r
```

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.

Read and write Commands include *data* bytes that are optional (can have a data length of zero).

```
Send write command c to resource ``r`` with ``n`` bytes of data ``d``

Send read command c to resource ``r`` and get ``n`` bytes of data ``d`` back
```

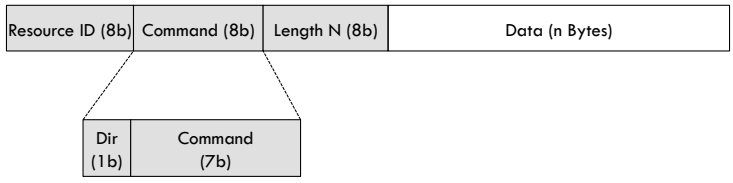

Fig. 2: Packet for control communications

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 xC interfaces.

Tasks *register* their resources and these get bound to the tasks' xC interface. When commands are received by the transport task they forwarded over the matching xC interface.

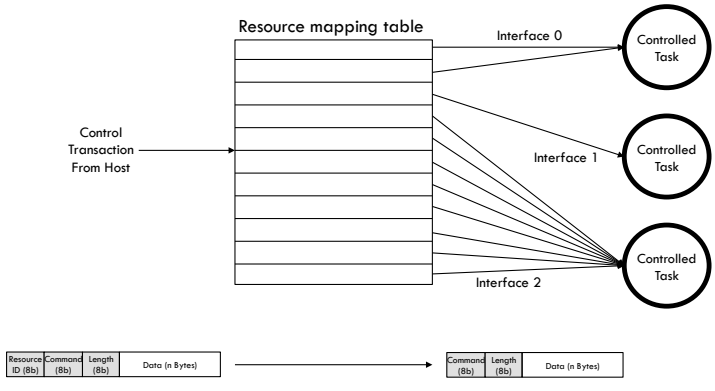

Fig. 3: Mapping between resource IDs and xC interfaces

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

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

The control library supports USB (device is USB device), I2C (device is I2C slave) and xSCOPE (device is target connected via xTAG debug adapter) as physical protocols. The maximum data packet size for each of the transport types is as follows:

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	256 Bytes	Arbitrary

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

3 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` functions can be accessed via the `device_control.h` header file, for example:

```
#include "device_control.h"
```

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 controlled resource.

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

Over I2C slave, the command is split into multiple I2C transactions:

```
process_i2c_write_transaction(reg, val)
process_i2c_write_transaction(reg, val)
process_i2c_write_transaction(reg, val)
process_i2c_write_transaction(reg, val) ==> case i.write_command(r, c, n, d[])
```

Over USB requests, the command is sent over a single USB request:

```
process_usb_set_request(header, data, len) ==> case i.write_command(r, c, n, d[])
```

It is the same for xSCOPE, the XMOS debug protocol:

```
process_xscope_upload(data, len) ==> case i.write_command(r, c, n, d[])
```

When the system starts, the transport task does an `init()` call, which asks all other tasks to register their resources:

```
init() ==> i.register_resources(r[])
```

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 the *Device side* and *Host side* sections for further details.

5 References

5.1 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

5.2 USB

- http://www.beyondlogic.org/usbnutshell/usb6.shtml

6 API

6.1 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

UNUSED_RES(x)

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

Returns

true if the read bit in the command is set

Returns

false if the read bit is not set

CONTROL_CMD_VALUE(c)

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

Parameters

- **c** – **[inout]** The transport command code converted to an application command code.

CONTROL_CMD_SET_READ(c)

Sets the read bit on a command code

Parameters

- **c** – **[inout]** The command code to set the read bit on.

CONTROL_CMD_SET_WRITE(c)

Clears the read bit on a command code

Parameters

- **c** – **[inout]** The command code to clear the read bit on.

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.

6.2 Device side

MAX_RESOURCES_PER_INTERFACE

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

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

6.3 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