Device Control Library

The Device Control Library provides the ability to configure and control an XMOS device from a host over a number of transport layers.

Features

- Simple read/write API
- Fully acknowledged protocol
- Includes different transports including I2C slave, USB requests, xSCOPE over xCONNECT and SPI slave
- Supports multiple resources per task

The table below shows combinations of host and transport mechanisms that are currently supported. Adding new transport layers and/or hosts is straightforward where the hardware supports it.

Host	I2C	USB	xSCOPE	SPI
PC / Windows		Yes	Yes	
PC / OSX		Yes	Yes	
Raspberry Pi / Linux	Yes	TBD		Yes
xCORE	Yes			

Table 1: Supported Device Control Library Transports

Typical resource usage

Less than 1KB of code space is needed for the target device, plus whatever the chosen transport layer library requires. The API is in the form of function calls, so no additional logical cores are consumed. I/O requirements also depend on which transport layer is used.

Software version and dependencies

This document pertains to version 3.2.1 of this library. It is known to work on version 14.3.2 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_xassert (>=3.0.1)
- lib_logging (>=2.1.1)

Related application notes

AN01034 - Using the Device Control Library over USB

1 Device Control Library

1.1 Introduction

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

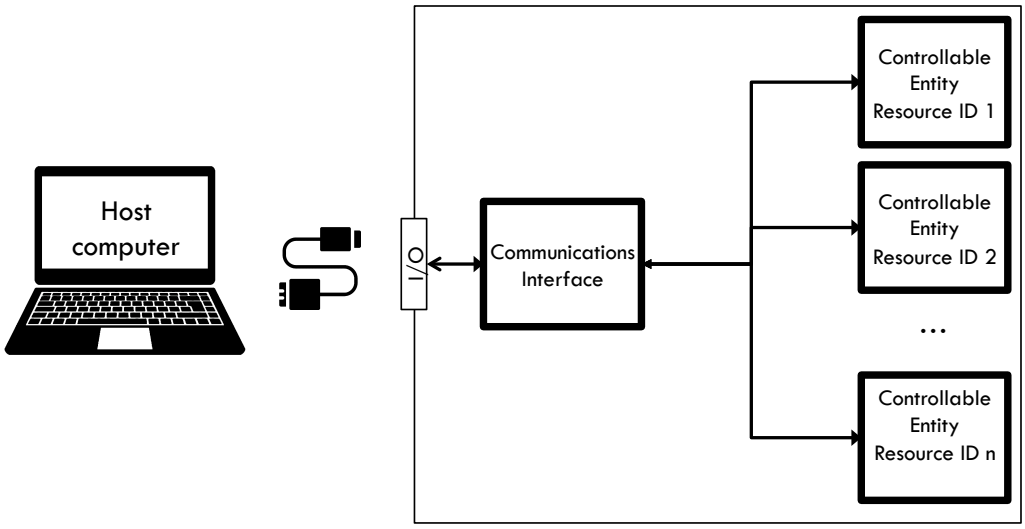

Figure 1: Logical view of lib_device_control

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.

1.2 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.

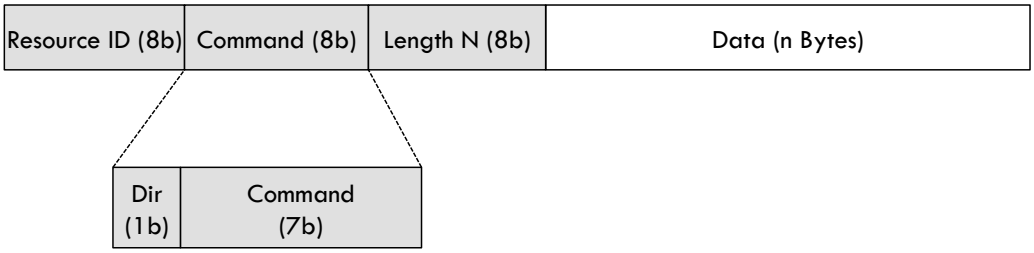

Figure 2: Packet for control communications

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

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.

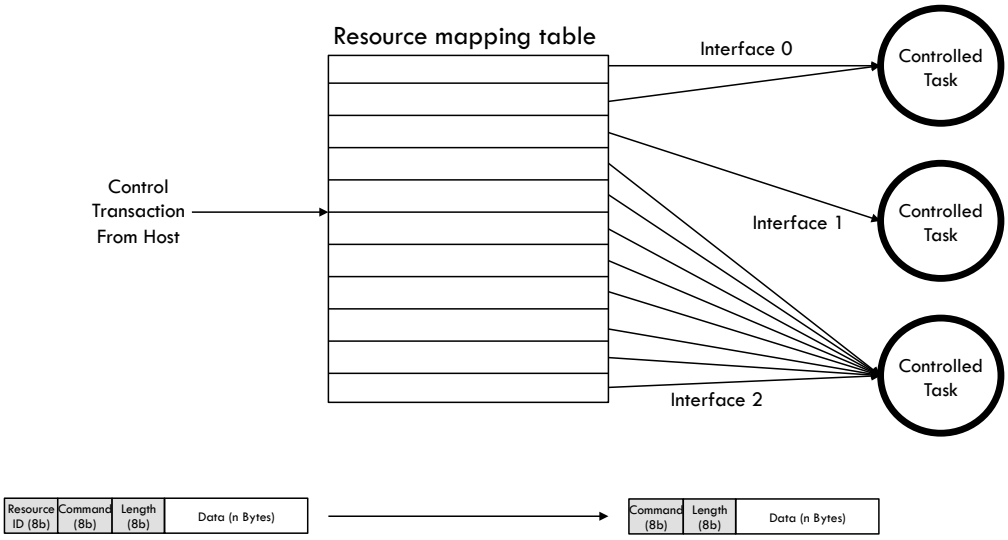

Figure 3: Mapping between resource IDs and xC interfaces

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 trasport types is as follows:

Transort	Data length	Limitation
I2C	253 Bytes	Arbitrary
USB	64 Bytes	USB control transfer specification
xSCOPE	256 Bytes	Arbitrary

Table 2: Maximum Data Length for Device Control Library Transports

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.

1.3 Usage

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 API—Device side section for further details.

1.4 References

1.4.1 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

1.4.2 USB

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

2 API—Device side

Type	control_ret_values
Description	This type enumerates the possible outcomes from a control transaction.
Values	CONTROL_SUCCESS CONTROL_REGISTRATION_FAILED CONTROL_BAD_COMMAND CONTROL_DATA_LENGTH_ERROR CONTROL_OTHER_TRANSPORT_ERROR CONTROL_ERROR

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

Continued on next page

Type	control (continued)
Functions	

Function	register_resources
Description	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.
Type	`void register_resources( control_resid_t resources[MAX_RESOURCES_PER_INTERFACE], unsigned &num_resources)`
Parameters	`resources` Array of resource IDs of size MAX_RESOURCES_PER_INTERFACE `num_resources` Number of resources populated within the resources[] table
Returns	void

Continued on next page

Type	control (continued)

Function	write_command
Description	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.
Type	`control_ret_t write_command(control_resid_t resid, control_cmd_t cmd, const uint8_t payload[payload_len], unsigned 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 indiciates 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

Continued on next page

Type	control (continued)

Function	read_command
Description	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.
Type	`control_ret_t read_command(control_resid_t resid, control_cmd_t cmd, uint8_t payload[payload_len], unsigned 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 indiciates 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

Function	control_init
Description	Initiaize the control library. Clears resource table to ensure nothing is registered.
Type	`control_ret_t control_init(void)`
Returns	Whether the initialization was successful or not

Function	control_register_resources
Description	Sends a request to the application to register controllable resources.
Type	`control_ret_t control_register_resources(client interface control i[n], unsigned n)`

Continued on next page

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	

Function	**control_process_i2c_write_start**
Description	Inform the control library that an I2C slave write has started. Called from I2C callback API.
Type	`control_ret_t` `control_process_i2c_write_start(client interface control i[])`
Parameters	`i` Array of interfaces used to communicate with controllable entities
Returns	Whether the write start was successful or not

Function	**control_process_i2c_read_start**
Description	Inform the control library that an I2C slave read has started. Called from I2C callback API.
Type	`control_ret_t` `control_process_i2c_read_start(client interface control i[])`
Parameters	`i` Array of interfaces used to communicate with controllable entities
Returns	Whether the read start was successful or not

Function	**control_process_i2c_write_data**
Description	Inform the control library that an I2C slave write has occured. Called from I2C callback API.

Type	`control_ret_t`
	`control_process_i2c_write_data(const uint8_t data,`
	`  client interface control i[])`

Parameters	`data` Array of byte data to be passed to the device `i` Array of interfaces used to communicate with controllable entities

Continued on next page

Returns	Whether the write was successful or not

Function	**control_process_i2c_read_data**
Description	Inform the control library that an I2C slave read has occured. Called from I2C callback API.

Type	`control_ret_t`
	`control_process_i2c_read_data(uint8_t &data,`
	`client interface control i[])`

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

Function	**control_process_i2c_stop**
Description	Inform the control library that an I2C transaction has stopped. Called from I2C callback API.
Type	`control_ret_t` `control_process_i2c_stop(client interface control i[])`
Parameters	`i` Array of interfaces used to communicate with controllable entities
Returns	Whether the stop was successful or not

Function	**control_process_usb_set_request**
Description	Inform the control library that a USB set (write) has occured. Called from USB EP0 handler.

Type	`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[])`

Continued on next page

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

Function	**control_process_usb_get_request**
Description	Inform the control library that a USB get (read) has occured. Called from USB EP0 handler.

Type	`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[])`

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

Function	**control_process_xscope_upload**
Description	Inform the control library that an xscope transfer has occured. 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.

Continued on next page

Type	`control_ret_t`
	`control_process_xscope_upload(uint8_t buf[],`
	`unsigned buf_size,`
	`unsigned length_in,`
	`unsigned &length_out,`
	`client interface control i[])`

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

3 API - Host side

Function	control_init_xscope
Description	Initialize the xscope host interface.

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

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

Function	control_cleanup_xscope
Description	Shutdown the xscope host interface.
Type	`control_ret_t control_cleanup_xscope(void)`
Returns	Whether the shutdown was successful or not

Function	control_init_i2c
Description	Initialize the I2C host (master) interface.
Type	`control_ret_t` `control_init_i2c(unsigned char i2c_slave_address)`
Parameters	`i2c_slave_address` I2C address of the slave (controlled device)
Returns	Whether the initialization was successful or not

Function	control_cleanup_i2c
Description	Shutdown the I2C host (master) interface connection.
Type	`control_ret_t control_cleanup_i2c(void)`
Returns	Whether the shutdown was successful or not

Function	control_init_usb
Description	Initialize the USB host interface.
Type	`control_ret_t control_init_usb(int vendor_id, int product_id, int interface_num)`
Parameters	`vendor_id` Vendor ID of controlled USB device `product_id` Product ID of controlled USB device `interface_num` USB Control interface number of controlled device
Returns	Whether the initialization was successful or not

Function	control_query_version
Description	Checks to see that the version of control library in the device is the same as the host.
Type	`control_ret_t control_query_version(control_version_t *version)`
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

Function	control_write_command
Description	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.
Type	`control_ret_t control_write_command(control_resid_t resid, control_cmd_t cmd, const uint8_t payload[], size_t payload_len)`

Continued on next page

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 indiciates 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

Function	**control_read_command**
Description	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.

Type	`control_ret_t`
	`control_read_command(control_resid_t resid,`
	`control_cmd_t cmd,`
	`uint8_t payload[],`
	`size_t 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 indiciates 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

APPENDIX A - Known Issues

- New installations of Windows 10 Anniversary (1607) and will not install the USB control driver without disabling attestation signing checks (lib_device_control #40)

APPENDIX B - Device control library change log

B.1 3.2.1

- Fix an issue on Windows where xSCOPE connection hangs on Windows (#17871)
- Fix an issue on Windows where first xSCOPE connection succeeds, but subsequent ones fail with "socket reply error" (#47)

B.2 3.2.0

- Updated XSCOPE and USB protocols for host applications
- Improved error messages in host applications
- Dummy LED OEN port in example applications
- Document Windows 10 attestation signing of libusb driver

B.3 3.1.1

- Use Vocal Fusion board XN file in xSCOPE and USB examples

B.4 3.1.0

- Add SPI support for Raspberry Pi host
- No longer down-shift I2C address on Raspberry Pi host

B.5 3.0.1

- Fixed incorrectly returned read data in xSCOPE example host code

B.6 3.0.0

- Replace xSCOPE and USB size limits in public API by runtime errors
- xSCOPE API change - buffer type from 64 words to 256 bytes
- Windows build fixes
- xTIMEcomposer project files for AN01034 and xSCOPE examples
- Documentation updates

B.7 2.0.2

- Added AN01034 application note based around USB transport example and xCORE Array Microphone board
- Documentation updates
- Increased test coverage

B.8 2.0.1

- Update XE232 XN file in I2C host example for tools version 14.2 (compute nodes numbered 0 and 2 rather than 0 and 1)

B.9 2.0.0

- Added the ability to select USB interface (Allows control from Windows)

B.10 1.0.0

- Initial version.
- Changes to dependencies:
  - lib_logging: Added dependency 2.1.0
  - lib_xassert: Added dependency 2.0.1