Developers guide

How INDIGO bus API looks like?

The core technology of INDIGO is INDIGO bus. It defines INDIGO properties, items, related types and function prototypes for bus itself and client and device driver.

INDIGO property represents a physical feature of the device (e.g. coordinates of a slew target of the telescope mount or status of camera cooler) or operation on the device (e.g. slew to the target or trigger camera exposure). It is a container for specified number items of the same type (e.g. RA and Dec part of equatorial coordinates). It is modelled according to the property in legacy INDI protocol.

Each property is related to a particular device, has the name, human readable group and label or description, state, type, permission, switch rule (optional for switch messages) and version (to allow mapping between different names or different behaviour in different versions of the standard).

The state of a property can be

- “idle” if property is passive or unused,
- “ok” if property is in correct state or if operation on property was successful,
- “busy” if property is transient state or if operation on property is pending and
- “alert” if property is in incorrect state or if operation on property failed.

The type of a property and property item can be

- “text” for strings of limited width,
- “number” for float numbers with defined min, max values and increment,
- “switch” for logical values representing “on” and “off” state,
- “light” for status values with four possible values “idle”, “ok”, “busy” and “alert” and
- “blob” for binary data of any type and any length.

The permission of a property can be “read only”, “read write” and “write only” with obvious meaning.

The switch property can have also switch rule defining dependencies between values of its items. Possible values are

- “one of many” and “at most one” for radio button group like behaviour and
- “any of many” for checkbox button group like behaviour.

Properties are defined in the “device” and used by “client”, pieces of code participating in bus communication and defined by structure containing among other pointers to callback functions executed by the bus if requested by other side.

In case of “device”, function pointers for the following events need to be defined:

- “attach” called when device is attached to the bus (e.g. on plug-in event),
- “enumerate properties” called when client request the list of available properties of the device (the list can actually differ according of the state of the device),
- “update property” called when client request the change of particular property (e.g. to set target temperature of CCD) and
- “detach” called when device is detached from the bus (e.g. on un-plug event).

In case of “client”, function pointers for the following events need to be defined:

- “attach” called when client is attached to the bus (e.g. handler process for remote client is created),
- “define property” called when device broadcast the definition of a new property (e.g. in reply to “enumerate properties” request or if connection state of device changed),
- “update property” called when device broadcast the value change of a property (e.g. if current temperature of CCD changed),
- “delete property” called when device broadcast removal of a property (e.g. if device is disconnected or its state changed).
- “send message” called when device broadcast a message (e.g. for debugging purposes).
- “detach” called when client is detached from the bus (e.g. when remote client closes the connection).

“Device” or “client” can be attached to or detached from the bus by calling function defined in bus API. Once connected, “device” can “define property”, “update property”, “delete property” or “send message”, while “client” can request “property enumeration” or “property update”. Property used in “property enumeration” request may have undefined device and/or property name, in this case all devices should response and/or all properties should be defined.

Among function pointers, both “device” and “client” structures contain pointer to device or client context (private data used by the code), result of last bus request and version (together with property version used for name transformation and/or specific request processing).

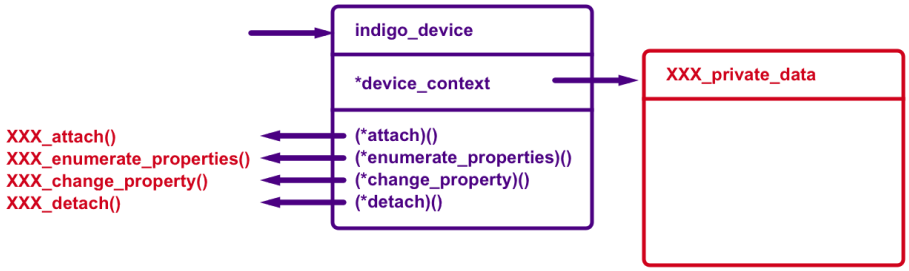
To learn more, look at source code in indigo_bus folder of INDIGO project on github.

How to write INDIGO driver?

Every device on the bus is identified by the structure of indigo_device type. Pointer to this structure is used in every bus API call. This structure should be allocated and initialized by driver supplied function and it depends on the nature of the driver, if it that function creates just one instance of the device (e.g. indigo_ccd_simulator() used by code managing the bus) or multiple devices (e.g. indigo_ccd_sx() used by USB hot-plug callback).

Before device is attached to the bus by indigo_attach_device(), device structure should be initialized by the pointers to driver entry points (some implementations of attach(), enumerate_properties(), change_property() and detach()).

Even if it is not mandatory, probably also some private data structure holding internal device state should be created at this stage and its address should be temporarily assigned to device_context attribute of device structure as shown on the following diagram.



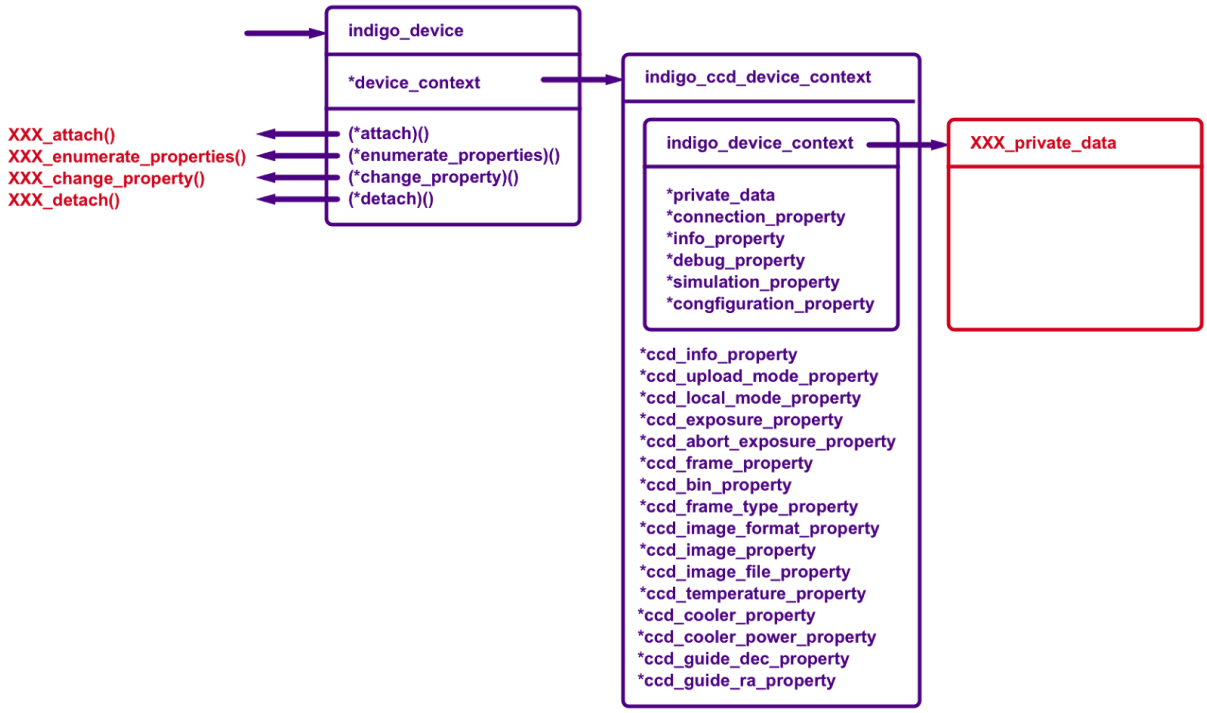
Once device is attached to the bus, attach() callback function is called. At this point driver must allocate and initialize device_context structure containing all device properties (and also some variable used internally by the framework). Most of the work can be done by calling corresponding function in base class implementation (e.g. indigo_ccd_device_attach() for CCD driver), driver should just assign private data pointer to private_data attribute of device context structure and setup initial state of device properties, if differ from default provided by base class function. attach() function should execute device's enumerate_properties() function to let clients know that device is ready (it is not propagated to remote clients over XML wire protocol unless they already send getProperties message).

Now device is ready to receive and process requests for property changes by change_property() callback function until detach() callback is called.

In change_property() implementation, driver should examine, if it is request to change the property managed by the device by indigo_property_match() function, and if it is, copy item values from request to the property structure managed by the device driver by indigo_property_copy_values() function. Once all driver specific processing is done, change_property() function should call corresponding function in base class implementation (e.g. indigo_ccd_device_change_property()).

In detach() implementation driver should free its private data structure and call corresponding function in base class implementation to free properties and device context (e.g. indigo_ccd_device_detach()).

On the following diagram is visualized the structure of data available for device driver for particular device. There are some macros defined in header files to make the work with the data easier. If any function in driver implementation has device parameter pointing to device structure, you can use DEVICE_CONTEXT macro to access device context structure, PRIVATE_DATA macro to access private data structure (you may want to redefine it to cast to correct structure as demonstrated in CCD simulator driver), macros with names like XXX_PROPERTY to access XXX property structure or XXX_YYY_ITEM to access item YYY in property XXX.



If driver should be used as a standalone executable, it also needs main() function. In its implementation bus should be started, device protocol adapter instantiated for standard input and output and attached to the bus, device instance instantiated and added or some kind of hot-plug detection should be started and finally XML wire protocol parser should be started. Both single and multiple device driver examples are demonstrated by CCD simulator and SX CCD driver main() function.

To learn more, look at source code in indigo_drivers folder of INDIGO project on github.