cROS:

A single thread C implementation of the ROS middleware

User Manual

Contents

Document History

cROS ver.	Doc. d/m/y	Doc. description	Doc. author
1.0-rc1	31/11/2018	First version with some sections	R.R. Carrillo, Aging in Vision and Action lab

Introduction

cROS is a single-thread C implementation of a ROS (Robot Operating System) client library.
ROS is middleware mainly intended for creating robot applications [1]. It provides an interface for communicating ROS nodes (which could be nodes in a TCP/IP network), including message passing, remote procedure call (RPC) and parameter-value sharing. This communication interface is implemented by several ROS client libraries [3], such as roscpp, rospy, roslisp and cROS, which can coexist and communicate among them since they implement the same protocol. ROS package also includes several software packages that make use of one or more ROS client libraries, for example for visualization or for package management.
This document describes the usage of cROS. Further information about cROS library can be found in [2].

ROS network

The ROS network is composed of ROS nodes (implemented by the client libraries), which can communicate between them. These ROS nodes can implement: a publisher of a topic (through predefined ROS messages), a subscriber to a topic, a service provider (a type of RPC), a service caller, a subscriber to a parameter, or several of them.
However, the ROS protocol requites that one special node is also present in the network: the ROS Master. This node provides the other nodes with a directory and registration service and a parameter dictionary through a XMLRPC interface. This Master node allows other ROS nodes to find out the address of the nodes that publish a specific topic or that provide the required service. This Master node also stores the last value of the parameter published by the nodes and update them when the value changes.

The Master node is usually created by means of the rosmaster package of ROS and launched using the roscore command. However, software such as the ROS toolbox of MATLAB 2015a (and later) also is able to create a ROS Master (and other nodes), which can be used by a ROS network.

When a topic publisher is created it registers in the ROS Master indicating the topic that it provides. When a topic subscriber is created it also registers in the Master indicating the topic to which it subscribes, and the Master replies with the publishers that publish this topic as illustrated in Figure 1. Then the subscriber contacts the publisher asking for the network address and port for the message reception connection using XMLRPC. Then the subscriber establishes this connection, exchanges information about the message format and transmission mode and start receiving the messages on the required topic (using TCPROS protocol [4]). When the publisher and subscriber nodes are destroyed they unregister from the ROS master.

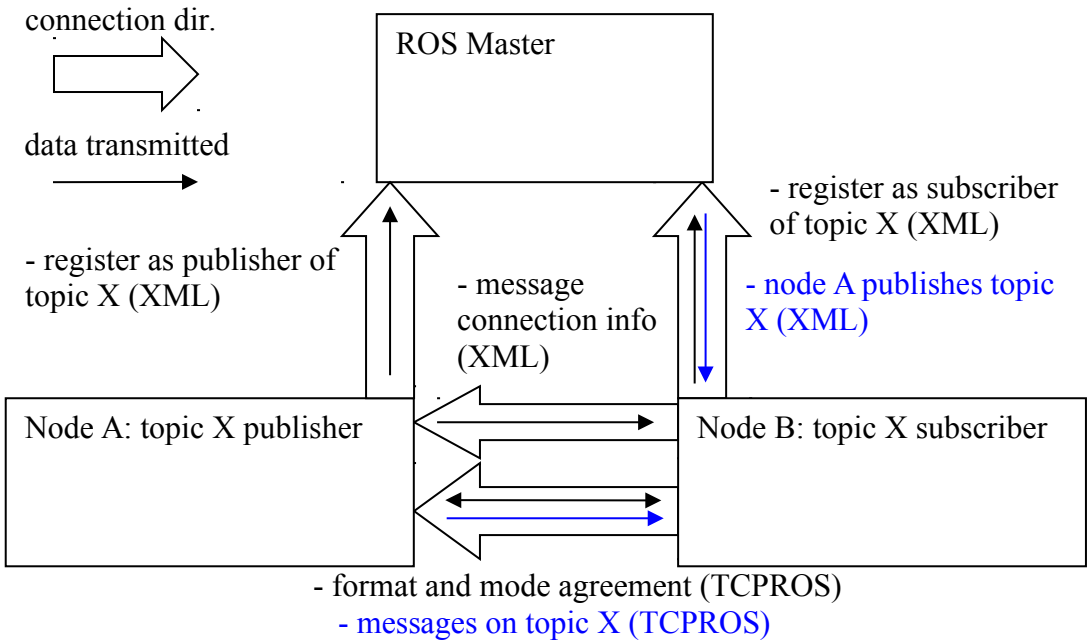

Figure 1: topic publisher/subscriber communication

When a service provider is created it registers in the ROS Master indicating the service that it provides. Then when a service client wants to make a service call it asks the Master what node provides this service. Then the caller establishes a connection to the provider, exchanges information about the service input and output format and transmission mode, sends the service request and receives the service response as illustrated in Figure 2 (using TCPROS protocol). When the provider is destroyed it unregisters from the ROS master.

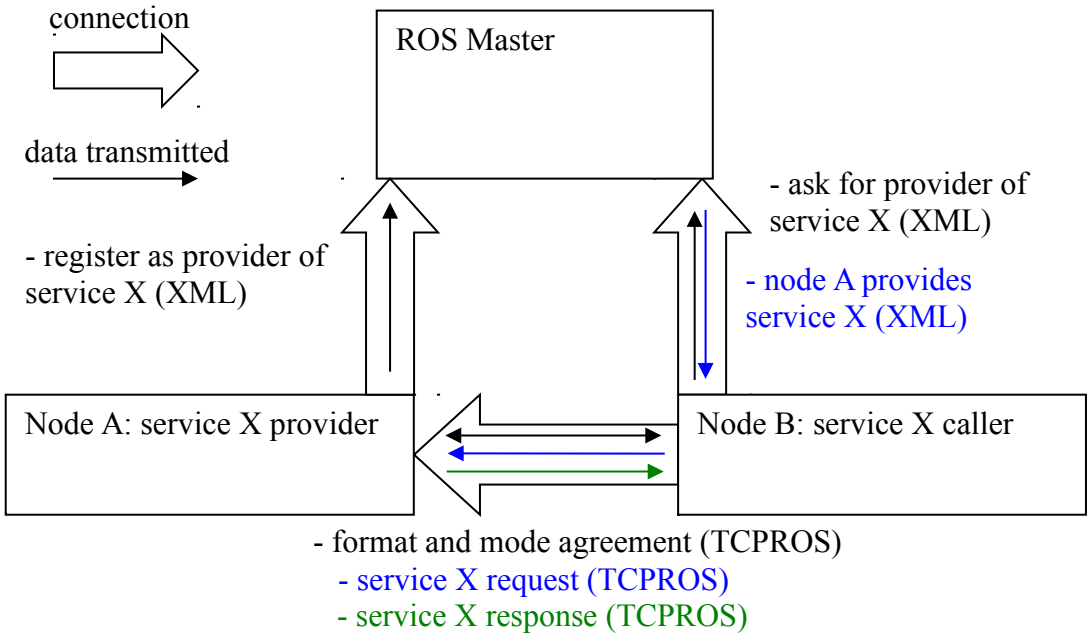

Figure 2: service provider/caller communication

When a parameter subscriber is created it registers in the ROS Master indicating the parameter about it is interested. Then when a node updates this parameter the Master

contacts the subscriber indicating the new parameter value as illustrated in Figure 3 (using XMLRPC protocol). When the subscriber is destroyed it unregisters from the ROS master.

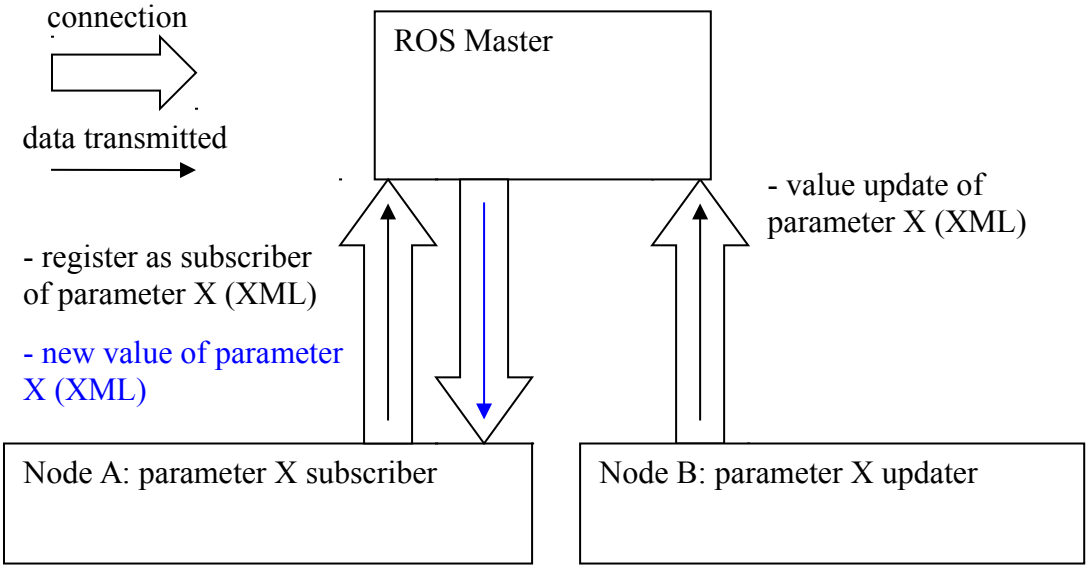

Figure 3: parameter updater/subscriber communication

cROS features

cROS is a ROS client-library implemented in C. The ROS node created by this library can be run by single thread thanks to its internal non-blocking read/write operations.
Can ROS be used without actually installing a ROS release?
cROS (as the other ROS client libraries) requires a ROS Master. This Master is not implemented by cROS, so other software must be run in the network to create the ROS Master. This software can be the rosmaster package included in a ROS release or it can be other application software such as MATLAB. However, if you do not want to install any of this software, the rosmaster package can be just downloaded and run directly by the Python interpreter. This means that you can have a ROS network running without actually installing a ROS release.

The key features of cROS are:

- It communicates with a ROS network by creating a ROS node that can act concurrently as several topic publishers, topic subscribers, service providers, service callers, parameter subscriber and parameter updaters.
- It loads and parses service and message text definition files (.msg and .srv) in execution time (no previous message or header creation is required).
- It can be compiled and executed in Linux and OS X (it has been tested with gcc). Only cmake is required if you want to use the provided compilation method.
- It is a dependence-free library; it means that no other third-party libraries have to be installed in order to compile it.
- It is a single-thread lightweight implementation.
- It supports call-back functions and polling for topic messages.

- It is compatible with the ROS network of MATLAB Robotics System Toolbox.

Compiling

cROS was initially designed to be compiled and executed in Linux, however it can also be used in Unix-based operating systems such as OS X.

Before compiling cROS the following packages must be installed:
- GNU C compiler and build essential (for make tool).
- cmake (in case that you want to use the provided compilation method).

To compile cROS, just execute the following command from the project directory:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

After compiling, in the `build/lib/` subdirectory the cROS library file `libcros.a` is found. Several executable files corresponding to several sample programs should be found in `build/bin/` subdirectory.

Sample programs

A set of test programs is provided to illustrate the cROS usage and test its operation.
The operation of most of these programs is documented in their source files.
These programs should be executed from the `build/bin/` subdirectory. They are:

Program filename	Description
`talker` and `listener`	They can be executed simultaneously to test the transmission of topic messages and service calls
`parameters-test`	It can be used together with `rosparam` command-line or MATLAB to test the parameter update and subscription.
`api-test`	It shows the use of top-level cROS API functions.
`raw-api-test`	It shows the use of internal cROS functions.
`ros-i-trajectory-test`	It creates a topic subscriber node and monitors the value of some of the message fields.

Project directories

The following directories exist inside the cROS project:

`build/` : The files resultant from the compilation are stored here.
`include/` : cROS header files are stored here.
`src/` : cROS source-code files are stored here.

`resources/` : Some test source-code, commands and definition files that can be used with a ROS release.
`samples/` : Source code and message and service definition files of test programs.

cROS interface overview

The main object of cROS is the ROS node (`CrosNode`). This node can implement a variable number of subscribers, publishers. For example, each subscriber subscribes to one topic. The maximum number of each one if defined a corresponding macro (defined in `cros_node.h`) in as illustrated in Figure 4.

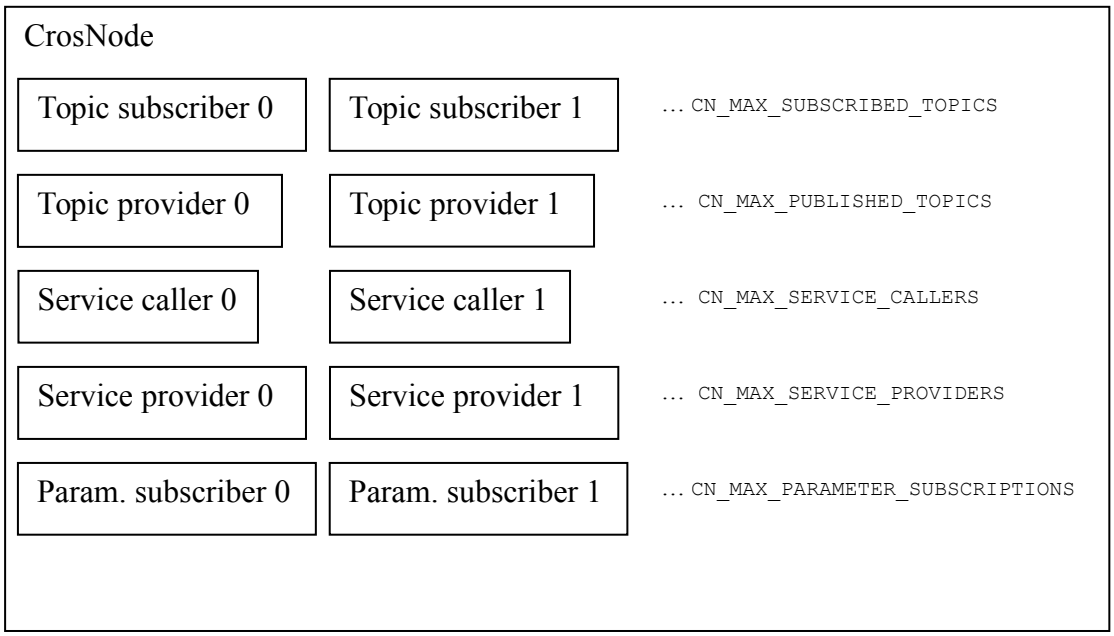

Figure 4: CrosNode in cROS

The cROS node is created by the `cRosNodeCreate()` function. This node is initially empty and must be filled with subscribers, publishers, etc. (we will call these the node roles). For this purpose cROS provide the functions:

- `cRosApiRegisterSubscriber()` / `cRosApiUnregisterSubscriber()`
- `cRosApiRegisterPublisher()` / `cRosApiUnregisterPublisher()`
- `cRosApiRegisterServiceCaller()` / `cRosApiReleaseServiceCaller()`
- `cRosApiRegisterServiceProvider()` / `cRosApiUnregisterServiceProvider()`
- `cRosApiSubscribeParam()` / `cRosApiUnsubscribeParam()`

When the `Subscribe` function or one of the `Register` functions is called the corresponding role is scheduled for registration in the ROS Master (except the `cRosApiRegisterServiceCaller()`, which is not registered in the Master). When the `Unsubscribe` function or one of the `Unregister` functions is called the corresponding role is scheduled for unregistration. When this unregistration completes the role is automatically released and a new one can occupy its position. The service callers do not need registered or unregistered from ROS Master, so the `cRosApiReleaseServiceCaller()` function can be called directly.

To update parameter values (in ROS Master) it is not necessary to create a role in the ROS node, the function `cRosApiSetParam()` can be used.
For a description of the parameter of these functions, please refer to the API section.

Callback functions

All the role-creation functions mentioned in the previous section have an optional `callback` parameter. This parameter allows the user to specify a user-defined function that will be called by cROS.
A topic subscriber will call its callback function when a new message about its topic is received, passing the received message to the function.
A topic publisher will call its callback function when a new message must be sent. So the function must provide this new message.
A service caller will call its callback function when a scheduled service call must be executed. So the function must provide the service call parameters (request). This function is also called when the service response is received. So this function must also process the service response.
A service provider will call its callback function when a service call is received. So the function must process the received service request and generate the corresponding service response.
A parameter subscriber will call its callback function when the status of the role has changed, for example when the parameter has been updated. So the function must process the new parameter value.
When callback functions are used the main parts of the program can be the followings:

- cROS node creation (`cRosNodeCreate()`)
- Roles creation (subscribers, providers, etc.)
- cROS node execution (`cRosNodeStart()`)
- cROS node destruction (`cRosNodeDestroy()`)

During `cRosNodeStart()` the execution the calling thread is blocked, so no other functions can be called meanwhile (apart from the callback functions that are called by cROS).
When using the callback mechanism for a topic publisher (or a service caller) the message publications (or service calls) are sent at a constant frequency specified when creating the node role.
The callback functions constitute one of the interfaces provided by cROS to pass data between cROS and the user software application. The other option is using polling.

Polling

The topic subscriber role support polling. So when a new message about a subscribed topic is received it is stored in a queue (of length `MAX_QUEUE_LEN` defined in `cros_message_queue.h`). When the function `cRosNodeReceiveTopicMsg()` is called the oldest message in the queue will be obtained. In case of queue overflow the oldest message is discarded.
When polling is used the general structure of the program can be the followings:

- cROS node creation (`cRosNodeCreate()`)
- Roles creation (subscribers, providers, etc.)

- Program main loop:
    - Single cROS node execution (`cRosNodeDoEventsLoop()`)
    - Polling (`cRosNodeReceiveTopicMsg()`)
    - Other user code
- cROS node destruction (`cRosNodeDestroy()`)

When using polling the `callback` parameter of the role-creation functions can be `NULL`. However, callback functions and polling can be used at the same time.
While this function is being executed the cROS node is run normally so that during this time new inputs are processed and planned outputs are generated.

Immediate message sending

The immediate message sending allows the user to send messages at arbitrary times (instead of being sent at a constant frequency, which is the case of just using callback functions). Calling the `cRosNodeSendTopicMsg()` function triggers the sending of a topic message. If this function is called several times and the previous message has not sent yet when it is sent, the specified message is stored in a queue (of length `MAX_QUEUE_LEN` defined in `cros_message_queue.h`). If the queue is full when a new message is tried to be inserted, `cRosNodeSendTopicMsg()` function will block during the specified timeout. During this blocking time, the cROS node is run normally: new inputs are processed and planned outputs are generated.
When immediate sending is required the general structure of the program can be similar to the one used for polling, so the user code can call the immediate sending functions.

Immediate service calling

The immediate service calling allows the user to make service calls at arbitrary times (instead of being called at a constant frequency, which is the case of just using callback functions). The `cRosNodeServiceCall()` function performs an immediate service call. This function blocks the thread execution until the service call response is received. During this blocking time, the cROS node is run normally by this function.
When immediate calling is required the general structure of the program can be similar to the one used for polling, so the user code can call the immediate sending functions.

cROS Application Programming Interface

The main (most common) functions provided by cROS library are described here and classified according to their purpose.

cROS node management

```
CrosNode *cRosNodeCreate(char *node_name, char *node_host, char
      *roscore_host, unsigned short roscore_port, char
      *message_root_path)
```

This function creates a node of the ROS network, which is necessary to use most of the other cROS functions. This node will include automatically a publisher of the topic `/rosout` and two service providers (services `~get_loggers` and `~set_logger_level`).

Parameter name	Parameter description
`node_name`	Name of the ROS node being created. It must not contain spaces.
`node_host`	Host address which is assigned to the ROS node (an address of the local host).
`roscore_host`	Address of the ROS Master node.
`roscore_port`	Port in which the ROS Master is listening. It is usually 113311.
`message_root_path`	Path to the directory where the definition files are found.
Return value	Pointer to the created node or `NULL` on failure.

```
cRosErrCodePack cRosNodeStart(CrosNode *node, unsigned long
      time_out, unsigned char *exit_flag)
```

This function runs the cROS node during the specified time period (`time_out`). During this time the calling thread is blocked running the node. If during this time the variable `exit_flag` becomes different from zero, the function will exit in a short time. If an error occurs, the function will exit immediately and return the error codes.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`time_out`	Maximum time that the function will be running the node (if no other exit condition is met).
`exit_flag`	Pointer to variable that is monitored during the node running. The function will exit if this variable becomes different from 0.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

```
cRosErrCodePack cRosNodeDoEventsLoop(CrosNode *node, uint64_t
      timeout)
```

This function runs the cROS node during a single cycle. During the node running time the calling thread is blocked. If no error occurs and the node is waiting to receive data, the function can take up to `timeout` milliseconds to finish. During this cROS node cycle all the ports are checked for new inputs. If no input is available, this function may block until a new input is received, a planned output take place or `timeout` is reached.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`timeout`	Maximum time that the function will be running the node. If this parameter is set to 0, the function will take the minimum time to complete an entire node cycle.

Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

`cRosErrCodePack cRosNodeDestroy(CrosNode *node)`

This function will free all the resources allocated for the cROS node. Before this, all the node roles are unregistered from the ROS Master. So it performs a clean node destruction, otherwise error codes are returned.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

`cRosErrCodePack cRosApiRegisterSubscriber(CrosNode *node, const char *topic_name, const char *topic_type, SubscriberApiCallback callback, NodeStatusCallback status_callback, void *context, int tcp_nodelay, int *subidx_ptr)`

This function creates a subscriber role in the cROS node. This subscriber registers in ROS Master for the specified topic. When the topic is available the subscriber connects to the corresponding providers and receives messages on the topic.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`topic_name`	The topic to which to subscribe.
`topic_type`	Type of the topic to which to subscribe. It can be the path to a message definition file not including the filename extension.
`callback`	Pointer to a function of type `SubscriberApiCallback` that will be called each time a new message is received. Or `NULL` if only polling is used to get the received messages.
`status_callback`	Pointer to a function of type `NodeStatusCallback` that will be called each time the subscriber status changes (i.e. when the topic is being registered in ROS Master and when it has been unregistered). Or `NULL` if no status updates are needed.
`context`	Pointer that will be passed to `callback` and `status_callback` functions. This pointer can be used by the user to provide context information to the callback. This information could be used inside these functions to differentiate the calls made to this function or to provide values and variables to be used by the function.
`tcp_nodelay`	1 if the TCP_NODELAY flag should be activated on the subscriber TCPROS socket that will receive the messages to minimize the transmission delays. 0 for normal socket operation.
`subidx_ptr`	Pointer to an integer where the identifier of the created subscriber is stored. Or `NULL` if the identifier is not needed.

Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
CallbackResponse SubscriberApiCallback(cRosMessage *message,
      void* data_context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`message`	Message codifying the received topic message.
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterSubscriber` function.
Return value	Unsigned byte that must be set to 0 when the callback function succeeds. If it is set to 1, an error code is returned by the cROS node top function being executed

The type of the status callback function is:

```
void NodeStatusCallback(CrosNodeStatusUsr *status, void*
      data_context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`status`	Integer codifying the last state change. It can be: `CROS_STATUS_NONE`: value not used so far. `CROS_STATUS_SUBSCRIBER_UNREGISTERED`: Subscriber has been unregistered from ROS master.
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterSubscriber` function.

```
cRosErrCodePack cRosApiRegisterPublisher(CrosNode *node, const
      char *topic_name, const char *topic_type, int loop_period,
      PublisherApiCallback callback, NodeStatusCallback
      status_callback, void *context, int *pubidx_ptr)
```

This function creates a publisher role in the cROS node. This publisher registers in ROS Master the specified topic. The publisher provides the messages on the topic when a subscriber connects to this publisher. The message publications can be periodic or triggered by the user through a function call.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`topic_name`	The topic to publish.
`topic_type`	Type of the topic to publish. It can be the path to a message definition file (not including the filename extension).

`loop_period`	Period in milliseconds of the message publication: A new message will be published at this frequency. Or -1 if periodic sending must not be used.
`callback`	Pointer to a function of type `PublisherApiCallback` that will be called each time a new message must be send (if `loop_period` is positive). The callback function is in charge of populating this message. Or `NULL` if periodic message sending is not used; that is, only immediate message sending is used.
`status_callback`	Pointer to a function of type `NodeStatusCallback` that will be called each time the publisher status changes (i.e. when the publisher has been unregistered from the ROS Master). Or `NULL` if no status updates are needed.
`context`	Pointer that will be passed to `callback` and `status_callback` functions. This pointer can be used by the user to provide context information to the callback. This information could be used inside these functions to differentiate the calls made to these functions or to provide values and variables to be used inside the functions.
`pubidx_ptr`	Pointer to an integer where the identifier of the created publisher is stored. Or `NULL` if the identifier is not needed.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
CallbackResponse PublisherApiCallback(cRosMessage *message,
     void* data_context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`message`	Message containing the fields of the topic message that will be published after the calback function execution. The callback function must populate these fields
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterPublisher` function.
Return value	Unsigned byte indicating whether the topic message could be correctly populated. If it is set to 0, the specified message topic will be published. If it is set to 1, the message is not published and an error code is returned by the cROS node top function being executed.

The type of the status callback function is:

```
void NodeStatusCallback(CrosNodeStatusUsr *status, void*
     data_context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`status`	Integer codifying the last state change. It can be: `CROS_STATUS_NONE`: value not used so far. `CROS_STATUS_PUBLISHER_UNREGISTERED`: Publisher has been unregistered from ROS master.
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterPublisher` function.

```
cRosErrCodePack cRosApiRegisterServiceCaller(CrosNode *node,
      const char *service_name, const char *service_type, int
      loop_period, ServiceCallerApiCallback callback,
      NodeStatusCallback status_callback, void *context, int
      persistent, int tcp_nodelay, int *svcidx_ptr)
```

This function creates a service-caller role in the cROS node. This service caller connects to a node that provides the specified service and performs calls (periodically or directly triggered by the user through a function call).

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`service_name`	The service to call.
`service_type`	Type of the service to call. It can be the path to a service definition file (not including the filename extension).
`loop_period`	Period in milliseconds of the service call: The service will be called at this frequency. Or -1 if periodic calling must not be used.
`callback`	Pointer to a function of type `ServiceCallerApiCallback`, which will be executed two times for each service call (if `loop_period` is positive). First the function is called to get the message request populated (in this case its `call_resp_flag` parameter is set to 0) and then it is called to get the received message response (in this case `call_resp_flag` is set to 1). Or `NULL` if periodic service calls are not used; that is, only immediate calls are used.
`status_callback`	It is not used. It must be set to `NULL`.
`context`	Pointer that will be passed to the `callback` function. This pointer can be used by the user to provide context information to the callback. This information could be used inside this function to differentiate the calls made to this function or to provide values and variables to be used inside the function.
`persistent`	1 if the connection to the service provider must not be closed after a service call is completed. This allows much faster calls. 0 if a new connection is negotiated and created for each service call.
`tcp_nodelay`	1 if the TCP_NODELAY flag should be activated on the caller TCPROS socket that will send message requests and receive message responses to minimize the transmission delays. 0 for normal socket operation.

`svcidx_ptr`	Pointer to an integer where the identifier of the created caller is stored. Or `NULL` if the identifier is not needed.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
CallbackResponse cRosApiRegisterServiceCaller(cRosMessage
      *request, cRosMessage *response, int call_resp_flag, void*
      context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`request`	Message containing the fields required for the service request that will be sent after the calback function execution. If `call_resp_flag` parameter is 0, the callback function must populate these fields.
`response`	Message containing the fields included in the service response. If `call_resp_flag` parameter is 1, the fields of this message are populated with the received message response.
`call_resp_flag`	Integer set to 0 when the callback function must specify the service request (the service is going to be called). It is set to 1 when the callback can consult the received response (the service call has been completed).
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterServiceCaller` function.
Return value	Unsigned byte that must be set to 0 when the callback function succeeds. If it is set to 1, an error code is returned by the cROS node top function being executed, in this case, if `call_resp_flag` is also set to 1 the service call will not be sent.

```
cRosErrCodePack cRosApiRegisterServiceProvider(CrosNode *node,
      const char *service_name, const char *service_type,
      ServiceProviderApiCallback callback, NodeStatusCallback
      status_callback, void *context, int *svcidx_ptr)
```

This function creates a service-provider role in the cROS node. This provider registers the specified service in ROS Master. The provider serves the call of the specified service when a caller connects to it.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`service_name`	The service to provide.
`service_type`	Type of the service to provide. It can be the path to a service definition file (not including the filename extension).

`callback`	Pointer to a function of type `ServiceProviderApiCallback` that will be called each time a new call of this service is received. The callback function is in charge of get the message request and populating the message response.
`status_callback`	Pointer to a function of type `NodeStatusCallback` that will be called each time the provider status changes (i.e. when the provider has been unregistered from the ROS Master). Or `NULL` if no status updates are needed.
`context`	Pointer that will be passed to `callback` and `status_callback` functions. This pointer can be used by the user to provide context information to the callback. This information could be used inside these functions to differentiate the calls made to these functions or to provide values and variables to be used inside these functions.
`svcidx_ptr`	Pointer to an integer where the identifier of the created provider is stored. Or `NULL` if the identifier is not needed.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
CallbackResponse ServiceProviderApiCallback(cRosMessage
     *request, cRosMessage *response, void* data_context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`request`	Message codifying the received service request.
`response`	Message containing the fields required for the service response that will be sent after the calback function execution. The callback function must set the fields of this message.
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterServiceProvider` function.
Return value	Unsigned byte indicating whether the service response message could be correctly populated. If it is set to 0, the specified service response will be sent. If it is set to 1, the service response is not sent and an error code is returned by the cROS node top function being executed.

The type of the status callback function is:

```
void NodeStatusCallback(CrosNodeStatusUsr *status, void*
     data_context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description

`status`	Integer codifying the last state change. It can be: `CROS_STATUS_NONE`: value not used so far. `CROS_STATUS_SERVICE_PROVIDER_UNREGISTERED`: Service provider has been unregistered from ROS master.
`data_context`	Pointer passed as `context` parameter to the `cRosApiRegisterServiceProvider` function.

```
cRosErrCodePack cRosApiSubscribeParam(CrosNode *node, const char
      *key, NodeStatusCallback callback, void *context, int
      *paramsubidx_ptr)
```

This function creates a parameter-subscriber role in the cROS node. This subscriber registers in ROS Master, so when any parameter under the specified key changes its value the subscriber is warned.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`key`	The key of the parameter to which to subscribe (monitor).
`callback`	Pointer to a function of type `NodeStatusCallback` that will be called each time the parameter status changes (i.e. when the value of the parameter changes).
`context`	Pointer that will be passed to the `callback` function. This pointer can be used by the user to provide context information to the callback. This information could be used inside this function to differentiate the calls made to this function or to provide values and variables to be used inside this function.
`paramsubidx_ptr`	Pointer to an integer where the identifier of the created subscriber is stored. Or `NULL` if the identifier is not needed.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
void NodeStatusCallback(CrosNodeStatusUsr *status, void*
      context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description

<table>
<tr><td>status</td><td>Structure codifying the new state of the parameter. Its most relevant fields are:
<table>
<tr><th>Member name</th><th>Member description</th></tr>
<tr><td>state</td><td>Integer field codifying the nature of the last state change:
CROS_STATUS_NONE: value not used so far.
CROS_STATUS_PARAM_UNSUBSCRIBED: when the parameter subscriber has been unregistered from the ROS master.
CROS_STATUS_PARAM_SUBSCRIBED: when the parameter subscriber has been registered in the ROS master.
CROS_STATUS_PARAM_UPDATE: when a key of the subscribed parameter has changed.</td></tr>
<tr><td>parameter_key</td><td>String codifying a parameter key in case it has been updated. E.g. /testparam/x/</td></tr>
<tr><td>parameter_value</td><td>XmlrpcParam structure codifying the value of the key in case it has been updated.</td></tr>
</table></td></tr>
<tr><td>context</td><td>Pointer passed as context parameter to the cRosApiSetParam function.</td></tr>
</table>

cRosErrCodePack cRosApiUnregisterSubscriber(CrosNode *node, int subidx)

This function unregisters the specified topic subscriber from ROS Master and deletes the role in the cROS node.

Parameter name	Parameter description
node	Pointer to the cROS node.
subidx	Identifier of the subscriber to unregister and delete.
Return value	CROS_SUCCESS_ERR_PACK on success. Otherwise a pack containing the code(s) of the occurred error(s).

cRosErrCodePack cRosApiUnregisterPublisher(CrosNode *node, int pubidx)

This function unregisters the specified topic publisher from ROS Master and deletes the role in the cROS node.

Parameter name	Parameter description
node	Pointer to the cROS node.
pubidx	Identifier of the publisher to unregister and delete.
Return value	CROS_SUCCESS_ERR_PACK on success. Otherwise a pack containing the code(s) of the occurred error(s).

void cRosApiReleaseServiceCaller(CrosNode *node, int svcidx)

This function deletes the service-caller role in the cROS node (the service caller is not registered in ROS Master).

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`svcidx`	Identifier of the caller to be deleted.

```
cRosErrCodePack cRosApiUnregisterServiceProvider(CrosNode *node,
      int svcidx)
```

This function unregisters the specified service provider from ROS Master and deletes the role in the cROS node.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`svcidx`	Identifier of the provider to unregister and delete.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

```
cRosErrCodePack cRosApiUnsubscribeParam(CrosNode *node, int
      paramsubidx)
```

This function unregisters the specified parameter subscriber from ROS Master and deletes the role in the cROS node.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`paramsubidx`	Identifier of the subscriber to unregister and delete.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

Management of cROS node messages

```
void cRosMessageInit(cRosMessage *message)
```

This function initializes a new (uninitialized) message. This function does not allocate memory for the main message structure, so `message` must already point to an allocated (or static) message structure. When the message is not needed anymore `cRosMessageRelease` must be applied to the message in order to free its content.

Parameter name	Parameter description
`message`	Pointer to an uninitialized message.

Example

Creation of a message (partially) using static memory:

```
cRosMessage msg;
cRosMessageInit(&msg);
…
cRosMessageRelease(&msg);
```

void cRosMessageRelease(cRosMessage *message)

This function frees the memory used by the message content (uninitializes the message). This function must be called with messages initialized with `cRosMessageInit` when they are not needed anymore. After using this function the message can not be used unless `cRosMessageInit` is called again. This function does not free the memory of the main message structure, so it is mainly intended for static messages.

Parameter name	Parameter description
`message`	Pointer to an initialized message.

cRosMessage *cRosMessageNew()

This function creates (allocates and initializes) a new message. This function constitutes the dynamic-memory alternative to `cRosMessageInit`. When the message is not needed anymore `cRosMessageFree` must be applied to the message.

Parameter name	Parameter description
Return value	Pointer to the created message. Or `NULL` if an error occurred allocating the new message memory.

Example

Creation of a cROS message using only dynamic memory:

```
cRosMessage *msg_ptr;
msg_ptr = cRosMessageNew();
…
cRosMessageFree(msg_ptr);
```

void cRosMessageFree(cRosMessage *message)

This function frees all the memory used by the specified message (uninitializes the message and free is structure memory). This function must be called with messages created with `cRosMessageNew` when they are not needed anymore. After using this function the message can not be used.

Parameter name	Parameter description
`message`	Pointer to a created message.

void cRosMessageFieldsPrint(cRosMessage *msg, int n_indent)

This function prints in console all the fields (and their values) recursively in the specified message. The message printing can be indented, that is, the screen in which the message should be printed can be different from 0.

Parameter name	Parameter description
`message`	Pointer to an initialized message.
`n_indent`	Number of blank columns to leave in the screen on the left of the message.

`cRosMessageField *cRosMessageGetField(cRosMessage *msg , char *field)`

This function searches for the specified field in the message and returns a pointer to it. This function only searches in the top messages fields. If the searched field is inside other field (nested), a pointer to the parent field (message) must be obtained first.

Parameter name	Parameter description
`msg`	Pointer to an initialized message.
`field`	Pointer to a string containing the name of the message to be searched for.
Return value	Pointer to the specified field. Or `NULL` if the specified field was not found in the message.

Inside the `cRosMessageField` structure the union `data` contains the value of the field. The value of the field can be automatically casted to its corresponding type by accessing the corresponding union member.

Field type	Union member	Preferred access mode
int8_t	as_int8	`m->data.as_int8`
uint8_t	as_uint8	`m->data.as_uint8`
int16_t	as_int16	`m->data.as_int16`
uint16_t	as_uint16	`m->data.as_uint16`
int32_t	as_int32	`m->data.as_int32`
uint32_t	as_uint32	`m->data.as_uint32`
int64_t	as_int64	`m->data.as_int64`
uint64_t	as_uint64	`m->data.as_uint64`
flota	as_float32	`m->data.as_float32`
double	as_float64	`m->data.as_float64`
char *	as_string	`m->data.as_string` `cRosMessageSetFieldValueString`
cRosMessage *	as_msg	`m->data.as_msg`
int8_t *	as_int8_array	`cRosMessageFieldArrayAtInt8` `cRosMessageFieldArrayPushBackInt8`
uint8_t *	as_uint8_array	`cRosMessageFieldArrayAtUInt8` `cRosMessageFieldArrayPushBackUInt8`
int16_t *	as_int16_array	`cRosMessageFieldArrayAtInt16` `cRosMessageFieldArrayPushBackInt16`
uint16_t *	as_uint16_array	`cRosMessageFieldArrayAtUInt16` `cRosMessageFieldArrayPushBackUInt16`
int32_t *	as_int32_array	`cRosMessageFieldArrayAtInt32` `cRosMessageFieldArrayPushBackInt32`

uint32_t *	as_uint32_array	cRosMessageFieldArrayAtUInt32 cRosMessageFieldArrayPushBackUInt32
int64_t *	as_int64_array	cRosMessageFieldArrayAtInt64 cRosMessageFieldArrayPushBackInt64
uint64_t *	as_uint64_array	cRosMessageFieldArrayAtUInt64 cRosMessageFieldArrayPushBackUInt64
float *	as_float32_array	cRosMessageFieldArrayAtFloat32 cRosMessageFieldArrayPushBackFloat32
double *	as_float64_array	cRosMessageFieldArrayAtFloat64 cRosMessageFieldArrayPushBackFloat64
char **	as_string_array	cRosMessageFieldArrayAtStringGet cRosMessageFieldArrayAtStringSet cRosMessageFieldArrayPushBackString
cRosMessage **	as_msg_array	cRosMessageFieldArrayAtMsgGet cRosMessageFieldArrayAtMsgSet cRosMessageFieldArrayPushBackMsg cRosMessageFieldArrayRemoveLastMsg
void * (any arr.)	as_array	cRosMessageFieldArrayPushBackZero cRosMessageFieldArrayClear

Example

Obtaining the value of a nested message field “child_name” which is inside field “parent_name”:

```
cRosMessageField *parent_field, *child_field;
cRosMessage *parent_msg;
uint32_t child_value;

parent_field = cRosMessageGetField(message, "parent_name");
parent_msg = parent_field->data.as_msg;
child_field = cRosMessageGetField(parent_msg, "child_name");
child_value = child_field->data.as_int32;
```

```
int cRosMessageFieldArrayPushBackInt8(cRosMessageField *field,
      int8_t val)
int cRosMessageFieldArrayPushBackUInt8(cRosMessageField *field,
      uint8_t val)
int cRosMessageFieldArrayPushBackInt16(cRosMessageField *field,
      int16_t val)
int cRosMessageFieldArrayPushBackUInt16(cRosMessageField *field,
      uint16_t val)
int cRosMessageFieldArrayPushBackInt32(cRosMessageField *field,
      int32_t val)
int cRosMessageFieldArrayPushBackUInt32(cRosMessageField *field,
      uint32_t val)
int cRosMessageFieldArrayPushBackInt64(cRosMessageField *field,
      int64_t val)
int cRosMessageFieldArrayPushBackUInt64(cRosMessageField *field,
      uint64_t val)
int cRosMessageFieldArrayPushBackFloat32(cRosMessageField
      *field, float val)
int cRosMessageFieldArrayPushBackFloat64(cRosMessageField
      *field, double val)
```

```
int cRosMessageFieldArrayPushBackString(cRosMessageField *field,
      const char* val)
int cRosMessageFieldArrayPushBackMsg(cRosMessageField *field,
      cRosMessage* msg)
```

These functions add a new element at the end of the array.

Parameter name	Parameter description
`field`	Pointer to the field message codifying the array.
`val`	New element to add.
Return value	0 If the operation succeeded. Otherwise -1, indicating a memory allocation error or a field that is incompatible with the operation.

```
int8_t *cRosMessageFieldArrayAtInt8(cRosMessageField *field, int
      position)
int16_t *cRosMessageFieldArrayAtInt16(cRosMessageField *field,
      int position)
int32_t *cRosMessageFieldArrayAtInt32(cRosMessageField *field,
      int position)
int64_t *cRosMessageFieldArrayAtInt64(cRosMessageField *field,
      int position)
uint8_t *cRosMessageFieldArrayAtUInt8(cRosMessageField *field,
      int position)
uint16_t *cRosMessageFieldArrayAtUInt16(cRosMessageField *field,
      int position)
uint32_t *cRosMessageFieldArrayAtUInt32(cRosMessageField *field,
      int position)
uint64_t *cRosMessageFieldArrayAtUInt64(cRosMessageField *field,
      int position)
float *cRosMessageFieldArrayAtFloat32(cRosMessageField *field,
      int position)
double *cRosMessageFieldArrayAtFloat64(cRosMessageField *field,
      int position)
char *cRosMessageFieldArrayAtStringGet(cRosMessageField *field,
      int position)
cRosMessage *cRosMessageFieldArrayAtMsgGet(cRosMessageField
      *field, int position)
```

These functions return a pointer to the specified element in an array. This pointer can be used to read the element value or to modify it.

Parameter name	Parameter description
`field`	Pointer to the field message codifying the array.
`position`	Position of the element to access in the array. The first element is in position 0.
Return value	Pointer to the indicated element. Otherwise `NULL` indicating an invalid specified element position or array type.

```
int cRosMessageSetFieldValueString(cRosMessageField *field,
      const char *value)
```

This function changes the value of a string field.

Parameter name	Parameter description
field	Pointer to message field codifying the string.
value	Pointer to a string that will be copied in the field.
Return value	0 If the operation succeeded. Otherwise -1, indicating a memory allocation error or a field that is incompatible with the operation.

```
int cRosMessageFieldArrayAtStringSet (cRosMessageField *field,
      int position, const char* val)
int cRosMessageFieldArrayAtMsgSet(cRosMessageField *field, int
      position, cRosMessage* val)
```

These functions change the value of an array-field element.

Parameter name	Parameter description
field	Pointer to message field codifying the array.
position	Position of the element to modify in the array. The first element is in position 0.
value	Pointer to the element that will be copied in the specified position.
Return value	0 If the operation succeeded. Otherwise -1, indicating a memory allocation error or a field that is incompatible with the operation.

```
cRosMessage *cRosMessageFieldArrayRemoveLastMsg
      (cRosMessageField *field)
```

This function returns the last element of a message-array field. The element is removed from the array. The returned message must be freed after being used using the function `cRosMessageFree`.

Parameter name	Parameter description
field	Pointer to the message field codifying the message array.
Return value	Pointer to the last message of the array. Otherwise `NULL` indicating an empty array or invalid array type.

```
int cRosMessageFieldArrayPushBackZero(cRosMessageField *field,
      int n_field)
```

This function adds a new zero (or empty) element at the end of an array. This function is especially useful when dealing with arrays containing complex elements, such as messages since the new message included in the array contains all the fields corresponding to its type. In this way, the library user does not need to compose the message to insert a new element in the array.

Parameter name	Parameter description
field	Pointer to the message containing the array.
n_field	Number of the field in the message that codifies the array in which the new element must be added.

Return value	0 If the operation succeeded. Otherwise -1, indicating a memory allocation error, an invalid field number or a field that is incompatible with the operation.

`int cRosMessageFieldArrayClear(cRosMessageField *field)`

This function deletes all the elements in an array.

Parameter name	Parameter description
`field`	Pointer to message field codifying the array.
Return value	0 If the operation succeeded. Otherwise -1, indicating a field that is incompatible with the operation.

`cRosMessage *cRosApiCreatePublisherMessage(CrosNode *node, int pubidx)`

This function creates a message containing the same fields as the specified topic. This function is useful to create the message needed as parameter of the `cRosNodeSendTopicMsg` function.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`pubidx`	Identifier of the publisher. This number if provided by the `cRosApiRegisterPublisher` function.
Return value	Pointer to the created message. Or `NULL` if the specified identifier is invalid or some error occurred allocating the new message memory.

`cRosMessage *cRosApiCreateServiceCallerRequest(CrosNode *node, int svcidx)`

This function creates a message containing the same fields as the message used in the specified service request. This function is useful to create the message needed as parameter of the `cRosNodeServiceCall` function.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`svcidx`	Identifier of the service caller. This number if provided by the `cRosApiRegisterServiceCaller` function.
Return value	Pointer to the created message. Or `NULL` if the specified identifier is invalid or some error occurred allocating the new message memory.

Polling and immediate message sending / service calling

cRosErrCodePac cRosNodeReceiveTopicMsg(CrosNode *node, int subidx, cRosMessage *msg, unsigned char *buff_overflow, unsigned long time_out)

This function provides the library user with the oldest topic message received (that has not been provided yet by this function). When a new topic message is received by the node it is queued. This function removes the oldest message in the queue. The queue has a limited storage capacity, if messages are not dequeued often enough, a queue overflow may happen. If there is not messages in the queue, the function waits (it runs the main node loop) until a message is received or the maximum waiting time is reached.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`subidx`	Identifier of the topic subscriber. This number if provided by the `cRosApiRegisterSubscriber` function.
`msg`	Pointer to an initialized message in which the received service message will be stored.
`buff_overflow`	Pointer to a integer that be set to 0 if a message queue overflow has not happened since the last call to this function. Otherwise, it is set to 1.
`time_out`	Maximum time to wait for a new message milliseconds. If this value is 0, the function checks if there is a message available in the queue, but it does not wait in case the queue is empty.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

Example

It gets the last message received on the topic of the subscriber `subidx` of the node `node`. It is assumed that the node and the subscriber role are already created:

```
cRosMessage msg;
cRosErrCodePac err_cod;
cRosMessageInit(&msg);
err_cod = cRosNodeReceiveTopicMsg( node, subidx , &msg, NULL,
     5000);
if(err_cod == CROS_SUCCESS_ERR_PACK)
    cRosMessageFieldsPrint(&msg, 0);
else
    cRosPrintErrCodePack(err_cod, " cRosNodeReceiveTopicMsg()
     returned an error code");
cRosMessageFieldsPrint(&msg, 0);
cRosMessageRelease(&msg);
```

cRosErrCodePac cRosNodeSendTopicMsg(CrosNode *node, int pubidx, cRosMessage *msg, unsigned long time_out)

This function publishes the specified topic message. This function tries to immediately add a copy of the specified message to the message sending queue. However, the queue has a limited storage capacity, if there is no space for new messages in the queue, the

function waits (it runs the main node loop) until space is available or the maximum waiting time is reached.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`pubidx`	Identifier of the topic publisher. This number if provided by the `cRosApiRegisterPublisher` function.
`msg`	Pointer to the message that must be sent.
`time_out`	Maximum time to wait for space in the queue milliseconds (this parameter is relevant only in case that the queue is full). If this value is 0, the function checks if there is space in the queue, but it does not wait in case the queue is full.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

Example

It sends a message on the topic of the publisher `pubidx` of the node `node`. It is assumed that the node and the publisher role are already created:

```
cRosMessage msg;
cRosErrCodePac err_cod;
msg = cRosApiCreatePublisherMessage(node, pubidx);
// Popullate msg
err_cod = cRosNodeSendTopicMsg(node, pubidx, msg, 5000);
cRosMessageFree(msg);
```

`cRosErrCodePac cRosNodeServiceCall(CrosNode *node, int svcidx, cRosMessage *req_msg, cRosMessage *resp_msg, unsigned long time_out)`

This function performs a service call. This is, it sends the specified service request to the corresponding service provider and waits for the response.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`svcidx`	Identifier of the service caller. This number if provided by the `cRosApiRegisterServiceCaller` function.
`req_msg`	Pointer to the message codifying the service request. This message can be created with `cRosApiCreateServiceCallerRequest` function.
`resp_msg`	Pointer to an initialized message in which the received service response will be stored.
`time_out`	Maximum time to wait for the service response in milliseconds.
Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

Example

It performs a service call using the service caller `svcidx` of the node `node`. It is assumed that the node and the service caller role are already created:

```
cRosMessage *msg_req, msg_res;
cRosMessageField *a_field, *b_field;
cRosErrCodePac err_cod;
msg_req = cRosApiCreateServiceCallerRequest(node, svcidx);
cRosMessageInit(&msg_res);

a_field = cRosMessageGetField(msg_req, "a");
b_field = cRosMessageGetField(msg_req, "b");
a_field->data.as_int64 = -1;
b_field->data.as_int64 = 10;
cRosMessageFieldsPrint(msg_req, 0);

err_cod = cRosNodeServiceCall(node, svcidx, msg_req, &msg_res,
     5000);
if(err_cod == CROS_SUCCESS_ERR_PACK)
    cRosMessageFieldsPrint(&msg_res, 1);
else
    cRosPrintErrCodePack(err_cod, "cRosNodeServiceCall()
     returned an error code");

cRosMessageFree(msg_req);
cRosMessageRelease(&msg_res);
```

ROS Master functions

`cRosErrCodePac cRosApiSetParam(CrosNode *node, const char *key, XmlrpcParam *value, SetParamCallback callback, void *context, int *caller_id_ptr)`

This function sets a parameter in ROS master. If the parameter did not exit, it is created. Otherwise its value is updated.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`key`	String encoding the parameter name.
`value`	Structure encoding the value of all the keys in the specified parameter.
`callback`	Callback function that will be called when the master response is received, providing this response to the library user. This function may also be called in case of error. This parameter can be `NULL` if the ROS master response is not needed.
`context`	Pointer that will be passed to the `callback` function. This pointer can be used by the user to provide context information to the callback. This information could be used inside this function to differentiate the calls made to this function or to provide values and variables to be used inside this function.

caller_id_ptr	Pointer to an integer than will receive an identifier of the call being made to ROS master. This parameter can be NULL if this number if not needed.
Return value	CROS_SUCCESS_ERR_PACK on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
void SetParamCallback(int callid, SetParamResult *result, void
     *context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
callid	Call identifier provided through caller_id_ptr parameter.
result	Structure codifying the response obtained from ROS master: Member name / Member description: code — Integer field codifying the result code. It is normally 1. status — String codifying the status of the operation. For example "parameter /testparam set" if the parameter name is /testparam. ignore — Integer codifying the ignore flag. It is normally 0. See [5] for more information about the parameters.
context	Pointer passed as context parameter to the cRosApiSetParam function.

```
cRosErrCodePac cRosApiGetParamNames(CrosNode *node,
     GetParamNamesCallback callback, void *context, int
     *caller_id_ptr)
```

This function gets a list of parameters set in ROS master. The list of parameter is obtained through a callback function.

Parameter name	Parameter description
node	Pointer to the cROS node.
callback	Callback function that will be called when the master response is received, providing the parameter names to the library user. This function may also be called in case of error.
context	Pointer that will be passed to the callback function. This pointer can be used by the user to provide context information to the callback. This information could be used inside this function to differentiate the calls made to this function or to provide values and variables to be used inside this function.
caller_id_ptr	Pointer to an integer than will receive an identifier of the call being made. This parameter can be NULL if this number if not needed.

Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
void getParamNamesCallback(int callid, GetParamNamesResult
     *result, void *context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`callid`	Call identifier previously provided through `caller_id_ptr` parameter.
`result`	Structure codifying the response obtained from ROS master: Member name / Member description `code` — Integer field codifying the result code. It is normally 1. `status` — String codifying the status of the operation, i.e. "Parameter names". `parameter_names` — Pointer to an array of strings which contain the parameter names. `parameter_count` — Number of strings in the array See [5] for more information about the parameters.
`context`	Pointer passed as `context` parameter to the `cRosApiGetParam` function.

```
cRosErrCodePac cRosApiDeleteParam(CrosNode *node, const char
     *key, DeleteParamCallback callback, void *context, int
     *caller_id_ptr)
```

This function deletes a parameter (o parameter key) from the ROS master.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`key`	String encoding the name of the parameter (or key) to delete.
`callback`	Callback function that will be called when the master response is received, providing the operation result to the library user. This function may also be called in case of error. This parameter can be `NULL` if the ROS master response is not needed.
`context`	Pointer that will be passed to the `callback` function. This pointer can be used by the user to provide context information to the callback. This information could be used inside this function to differentiate the calls made to this function or to provide values and variables to be used inside this function.
`caller_id_ptr`	Pointer to an integer than will receive an identifier of the call being made. This parameter can be `NULL` if this number if not needed.

Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
void DeleteParamCallback(int callid, DeleteParamResult *result,
     void *context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`callid`	Call identifier previously provided through `caller_id_ptr` parameter.
`result`	Structure codifying the response obtained from ROS master: Member name: `code` — Integer field codifying the result code. It is normally 1. Member name: `status` — String codifying the status of the operation. For example "parameter /testparam deleted" if the deleted parameter is /testparam. Member name: `ignore` — Integer codifying the ignore flag. It is normally 0. See [5] for more information about the parameters.
`context`	Pointer passed as `context` parameter to the `cRosApiDeleteParam` function.

```
cRosErrCodePac cRosApiLookupService(CrosNode *node, const char
     *service, LookupServiceCallback callback, void *context,
     int *caller_id_ptr)
```

This function gets the provider of the specified service. The result is obtained through a callback function.

Parameter name	Parameter description
`node`	Pointer to the cROS node.
`service`	String codifying the fully-qualified name of the service
`callback`	Callback function that will be called when the master response is received, providing the service provider URL to the library user. This function may also be called in case of error.
`context`	Pointer that will be passed to the `callback` function. This pointer can be used by the user to provide context information to the callback. This information could be used inside this function to differentiate the calls made to this function or to provide values and variables to be used inside this function.
`caller_id_ptr`	Pointer to an integer than will receive an identifier of the call being made. This parameter can be `NULL` if this number if not needed.

Return value	`CROS_SUCCESS_ERR_PACK` on success. Otherwise a pack containing the code(s) of the occurred error(s).

The type of the callback function is:

```
void LookupServiceCallback (int callid, LookupServiceResult
     *result, void *context)
```

The meaning of the parameters of this function is:

Parameter name	Parameter description
`callid`	Call identifier previously provided through `caller_id_ptr` parameter.
`result`	Structure codifying the response obtained from ROS master: Member name / Member description: `code` — Integer field codifying the result code. It is normally 1. `status` — String codifying the status of the operation, i.e. "rosrpc URI: [rosrpc://hostname:port/]". `service_result` — Pointer to a string codifying the service URL (address and port). E.g. "rosrpc://hostname:port/" Where hostname and port are the hostname and port is the port of the service provider. See [5] for more information about these parameters.
`context`	Pointer passed as `context` parameter to the `cRosApiLookupService` function.

Management of cROS XMLRPC parameter structures

`void xmlrpcParamInit(XmlrpcParam *param)`

This function initializes a new (uninitialized) parameter structure. This function does not allocate memory for the main message structure, so `param` must already point to an allocated (or static) parameter structure. When the structure is not needed anymore `xmlrpcParamRelease` must be applied to the structure in order to free its content.

Parameter name	Parameter description
`param`	Pointer to an uninitialized parameter structure.

Example

Creation of a XMLRPX parameter structure (partially) using static memory:

```
XmlrpcParam param;
xmlrpcParamInit(&param);
…
xmlrpcParamRelease(&param);
```

void xmlrpcParamRelease(XmlrpcParam *param)

This function frees the memory used by the content of the parameter structure (uninitializes the structure). This function must be called with structures initialized with `xmlrpcParamInit` when they are not needed anymore. After using this function the structure can not be used unless `xmlrpcParamInit` is called again. This function does not free the memory of the main structure, so it is mainly intended for static messages.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure.

XmlrpcParam *xmlrpcParamNew()

This function creates (allocates and initializes) a new XMLRPC parameter structure. This function constitutes the dynamic-memory alternative to `xmlrpcParamInit`. When the message is not needed anymore `xmlrpcParamFree` must be applied to the structure.

Parameter name	Parameter description
Return value	Pointer to the created structure. Or `NULL` if an error occurred allocating the new structure memory.

Example

Creation of a XMLRPC parameter structure using dynamic memory:

```
XmlrpcParam *param_ptr;
param_ptr = xmlrpcParamNew();
…
xmlrpcParamFree(param_ptr);
```

void xmlrpcParamFree(XmlrpcParam *param)

This function frees all the memory used by the specified parameter structure (uninitializes the structure and free is memory). This function must be called with structures created with `xmlrpcParamNew` when they are not needed anymore. After using this function the structure can not be used.

Parameter name	Parameter description
`param`	Pointer to a created parameter structure.

void xmlrpcParamPrint(XmlrpcParam *param)

This function prints in console all the fields (their types and values) recursively in the specified parameter structure.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure.

```
void xmlrpcParamSetUnknown(XmlrpcParam *param)
void xmlrpcParamSetBool(XmlrpcParam *param, int val)
void xmlrpcParamSetInt(XmlrpcParam *param, int32_t val)
void xmlrpcParamSetDouble(XmlrpcParam *param, double val)
void xmlrpcParamSetString(XmlrpcParam *param, const char *val)
```

These functions set the data type of the specified XMLRPC parameter structure to a single element type. That is, one element of the specified data type can be stored.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure.
`val`	New value of the element.

```
void xmlrpcParamSetArray(XmlrpcParam *param)
```

This function sets the data type of the specified XMLRPC parameter structure to an array. That is, a list of elements of the same data type.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure.

```
void xmlrpcParamSetStruct(XmlrpcParam *param)
```

This function sets the data type of the specified XMLRPC parameter structure to a structure. That is, a container of different data types.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure.

```
int xmlrpcParamGetBool(XmlrpcParam *param)
int32_t xmlrpcParamGetInt(XmlrpcParam *param)
double xmlrpcParamGetDouble(XmlrpcParam *param)
char *xmlrpcParamGetString(XmlrpcParam *param)
```

These functions get the value of the specified XMLRPC parameter structure if the parameter structure contains a single element of the specified data type.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure which has the same type as the calling function.

```
XmlrpcParam *xmlrpcParamArrayGetParamAt(XmlrpcParam *param, int
      idx)
```

This function gets one element of the specified XMLRPC parameter structure. The type of the specified XMLRPC parameter structure must be array.

Parameter name	Parameter description
`param`	Pointer to an initialized parameter structure.

idx	Number of the element to get from the array. The first element is number 0.
Return value	Pointer to the parameter structure containing the specified element.

```
XmlrpcParam * xmlrpcParamStructGetParam(XmlrpcParam *param,
      const char *name)
```

This function gets one field of the specified XMLRPC parameter structure. The type of the specified XMLRPC parameter structure must be structure.

Parameter name	Parameter description
param	Pointer to an initialized parameter structure.
name	String containing the name of the structure field to get.
Return value	Pointer to the parameter structure containing the specified field.

```
XmlrpcParam *xmlrpcParamArrayPushBackBool(XmlrpcParam *param,
      int val)
XmlrpcParam *xmlrpcParamArrayPushBackInt(XmlrpcParam *param,
      int32_t val)
XmlrpcParam *xmlrpcParamArrayPushBackDouble(XmlrpcParam *param,
      double val)
XmlrpcParam *xmlrpcParamArrayPushBackString(XmlrpcParam *param,
      const char *val)
```

These functions add a new element to the end of the specified XMLRPC parameter structure. The data type of the specified parameter structure must be array.

Parameter name	Parameter description
param	Pointer to an initialized parameter structure.
val	Value of the new element.
Return value	Pointer to the parameter structure containing the new element, or NULL if the new element could not be added.

```
XmlrpcParam *xmlrpcParamArrayPushBackArray(XmlrpcParam *param)
```

This function adds a new empty (array) element to the end of the specified XMLRPC parameter structure. The data type of the specified parameter structure must be array.

Parameter name	Parameter description
param	Pointer to an initialized parameter structure.
Return value	Pointer to the parameter structure containing the new array, or NULL if this new element could not be added.

```
XmlrpcParam *xmlrpcParamArrayPushBackStruct(XmlrpcParam *param)
```

This function adds a new empty (structure) element to the end of the specified XMLRPC parameter structure. The data type of the specified parameter structure must be structure.

Parameter name	Parameter description
Param	Pointer to an initialized parameter structure.
Return value	Pointer to the parameter structure containing the new structure, or NULL if this new element could not be added.

```
XmlrpcParam *xmlrpcParamStructPushBackBool(XmlrpcParam *param,
      const char *name, int val)
XmlrpcParam *xmlrpcParamStructPushBackInt(XmlrpcParam *param,
      const char *name, int32_t val)
XmlrpcParam *xmlrpcParamStructPushBackDouble(XmlrpcParam *param,
      const char *name, double val)
XmlrpcParam *xmlrpcParamStructPushBackString(XmlrpcParam *param,
      const char *name, const char *val)
```

These functions add a new field to the specified XMLRPC parameter structure. The data type of the specified parameter structure must be structure.

Parameter name	Parameter description
param	Pointer to an initialized parameter structure.
name	String containing the name of the structure field to be added.
val	Value of the new field.
Return value	Pointer to the parameter structure containing the new field, or NULL if the new field could not be added.

```
XmlrpcParam *xmlrpcParamStructPushBackArray(XmlrpcParam *param,
      const char *name)
```

This function adds a new empty-array field to the specified XMLRPC parameter structure. The data type of the specified parameter structure must be structure.

Parameter name	Parameter description
param	Pointer to an initialized parameter structure.
name	String containing the name of the structure field to be added.
Return value	Pointer to the parameter structure containing the new field, or NULL if the new field could not be added.

```
XmlrpcParam *xmlrpcParamStructPushBackStruct(XmlrpcParam *param,
      const char *name)
```

This function adds a new empty-structure field to the specified XMLRPC parameter structure. The data type of the specified parameter structure must be structure.

Parameter name	Parameter description
Param	Pointer to an initialized parameter structure.
name	String containing the name of the structure field to be added.
Return value	Pointer to the parameter structure containing the new field, or NULL if the new field could not be added.

Error codes of cROS functions

Some simple cROS API functions can fail because of a single reason. In this type of functions we find some functions that just return an integer value to indicate the failure. This integer value is -1 is the function has failed. Some other functions of this same type return a pointer, and this pointer is `NULL` when they fail.
However, some complex cROS API functions can fail because of a variety of reasons. Some of them can encounter an error and try to continue, so several errors could occur during the function execution. So in order to report all the arisen errors, these complex functions of cROS return a `cRosErrCodePack` data type. This data type is an integer number that codifies a pack of errors, so it can codify several error codes or none. The values of these error codes (`cRosErrCode`) (stored in this pack) are global for all the functions, and can be translated into the corresponding error-reporting strings using cROS functions.
When `cRosErrCodePack` is empty its value is `CROS_SUCCESS_ERR_PACK`, so we can know if a function succeeded just by comparing the value that it returned. In case the function failed some cROS error functions are provided to extract the error codes from the pack or to print the corresponding error messages in console.

```
int cRosPrintErrCodePack(cRosErrCodePack err_cod_pack, const
      char *fmt_str, ...)
```

This function prints the error messages corresponding to all the error codes contained in the specified error code pack, preceded by the formatted specified string.

Parameter name	Parameter description
`err_cod_pack`	Error code pack whose error messages are print. It is usually returned by a failing function.
`fmt_str`	String that specifies how subsequent arguments are printed in console. It has the same format as `printf` function.
Return value	Number of characters written in the console.

Example

It checks if the function `cRosNodeStart` has failed, and if so, prints the corresponding error messages:

```
cRosErrCodePack err_cod;
err_cod = cRosNodeStart(node,CROS_INFINITE_TIMEOUT,&exit_flag);
if(err_cod != CROS_SUCCESS_ERR_PACK)
   cRosPrintErrCodePack(err_cod, "Error while running ROS
     node");
```

```
cRosErrCode cRosGetLastErrCode(cRosErrCodePack err_pack)
```

This function gets the last error code inserted in the specified error code pack.

Parameter name	Parameter description
`err_pack`	Error code pack whose last error code must be obtained.
Return value	The last error code of the pack or `CROS_NO_ERR` if no error is codified in the pack.

cRosErrCodePack cRosRemoveLastErrCode(cRosErrCodePack prev_err_pack)

This function removes the last error code inserted in the specified error code pack.

Parameter name	Parameter description
`prev_err_pack`	Error code pack whose last error code must be removed.
Return value	The new error code pack with the last code removed. `CROS_SUCCESS_ERR_PACK` is retuned if `prev_err_pack` is `CROS_SUCCESS_ERR_PACK`.

const char *cRosGetErrCodeStr(cRosErrCode err_code)

This function returns the error-message string corresponding to the specified error code.

Parameter name	Parameter description
`err_code`	Error code to be translated into error message.
Return value	Error message string corresponding to the specified code. If no message is found for the specified code, `NULL` is returned.

Example

Custom implementation of the `cRosPrintErrCodePack` function to print the error messages contained in `err_cod_pack`:

```
cRosErrCode curr_err_cod;
while((curr_err_cod=cRosGetLastErrCode(err_cod_pack)) !=
      CROS_NO_ERR)
{
  msg_str = cRosGetErrCodeStr(curr_err_cod);
  if(msg_str != NULL)
    printf("%s\n", curr_err_cod, msg_str);
  err_cod_pack = cRosRemoveLastErrCode(err_cod_pack);
}
```

References

1. http://wiki.ros.org/ROS/Introduction
2. https://rosindustrial.squarespace.com/s/cros_tutorial.pdf
3. http://wiki.ros.org/Client%20Libraries
4. http://wiki.ros.org/ROS/TCPROS
5. http://wiki.ros.org/ROS/Parameter%20Server%20API