ROS and struct script manual entry

new 5. Structs

Set of variables can be aggregated into structs, and thus transferred and stored as a single variable. This reflects the capability in other languages, but comes with a few restrictions that should be taken into account.

Structs can be obtained through multiple means:

1. Using the struct function
2. Using the make_anon_struct function
3. Doing an RPC call that returns a struct
4. Subscribing to a ROS2 topic

The struct function takes one or more named arguments, and each argument name becomes a member in the struct. All values must be initialized by value, and the type of the value cannot be changed subsequently.

Creating a struct

```
# Create a struct
myStruct = struct(identifier1 = 1, identifier2 = 2, myMember = "Hello structs")

# Reassign a member
myStruct.myMember = "Goodbye structs"

# use a member
myVar = myStruct.myMember

# Use the second member by index (identifier2)
myVar = myStruct[1]

# A nested struct, stored by value
myStruct = struct(myStructMember = struct(myMember = "Hi nested struct") )
```

In various cases, having non numeric values in a list structure is advantageous. For that purpose, structs can also be initialized using the make_anon_struct function.

Creating a struct

```
# Create a struct with 10 "Test" strings, and no member names
myStruct = make_anon_struct(10, "Test")

# Reassign a member
myStruct[2] = "The third test string"
```

Each member has no name and can only be accessed using the [] operator. Unlike lists, structs can contain non numeric values, and thus does not have algebraic operators defined on them.

amend 7. Remote procedure call by adding

If and only if a struct returned by a remote procedure call only has the members "x", "y", "z", "rx", "ry", "rz", and those members are integer or floating point values, the struct will be converted to a pose in URScript.

new 8. Robot operating system 2 (ROS2)

Robot operating system 2 (ROS2) is a set of libraries that can be used for communication between robots or parts thereof.

ROS2 works with the concepts of **publishers** and **subscribers**, who can, respectively, publish or subscribe to **topics**.
Each topic is associated with a **message type**, and the publisher and subscriber must agree on the message type for the topic for the communication to work.
In URScript, an example implementation sending messages between threads on the robot can demonstrate the concept, though the overhead means this is not as efficient/fast as other available means of communication such as global variables:

Creating a struct

```
thread publish():
  pub_handle = ros_publisher_factory(topic="test", msg_type="std_msgs/String")
  while(True):
    pub_handle.write(struct(data="Hello ROS2"))
    sleep(1)
  end
end
myPublisher = run publish()

thread subscribe():
  sub_handle = ros_subscriber_factory(topic="test", msg_type="std_msgs/String")
  while (True):
    value = sub_handle.read()
    textmsg(value.data)
  end
end
mySubscriber = run subscribe()
join mySubscriber # Avoid the main thread exiting
```

ros_publisher_factory is used to create a publisher handle that can be used to publish/**write** messages to the topic. In the above example, it publishes to the topic "test" with a message type from ROS2 standard messages (std_msgs) with the specific type String.
In order to use a message type, the corresponding dynamic library (in this case libstd_msgs__rosidl_typesupport_introspection_c.so) must be present on the robot, where it will be dynamically loaded when needed.
Looking at the message definition on ros2.org, the std_msgs/String message type must have a single member named data of the type string. The struct passed to the write method on the handle must have exactly the same members of the correct types.

ros_subscriber_factory is used to create a subscriber handle. The constraints on the message type from publishers also apply to subscribers, but the read method returns a struct corresponding to the message type, blocking until one is available.

Both handles have more methods defined on them, which are documented in the respective functions documentation in this script manual, and further information about ROS2 can be obtained at ros.org.

New functions

struct(...)

Create a struct

A nonempty set of named parameters are tied together in a struct with the name of the arguments being used to name the members of the struct, and the values defining the types and values of those members.

Parameters:

The struct function is unique in that it takes a variable number of named arguments. It is subject to the following constraints:

- Each name can only occur once in each struct.
- The value assigned to each member determines the type of the member, and that type cannot be change subsequently.
- If a members value is assigned to a struct, that struct is *copied* into the new struct. You cannot create pointers to other structs.

- If a member is assigned to an anonymous struct as created by **make_anon_struct**, that member can later be assigned a shorter anonymous struct of the same type, but it can never grow beyond its original size.

Example command: myVar = struct(member1 = "Hi structs", anotherMember = 1, member2 = anotherVariable)

make_list(n, number)

make a new list of length n with the initial value of each element given by "number"

Parameters:

n: The length of the list

number: The initial value of each of the members of the list

Example command: make_list(10, 5.5)

make_anon_struct(n, value)

make a new anonymous struct of length n with the initial value of each element given by "value".
The types of the values in the anonymous struct are constrained to be the same, and as its members are unnamed, the [] operator must be used to access the members.

Tha capacity of the anonymous struct is set at initialization, and can never grow, but shorter anonymous structs can be assigned to it, providing the contained types are identical. E.g.

myVar = make_anon_struct(10, 1) can be followed by myVar = make_anon_struct(5, 1) but **not** myVar = make_anon_struct(11, 1)

Parameters:

n: The length of the anonymous struct

value: The initial value of each of the members of the struct. Note that value is not constrained to be a number.

Example command: make_anon_struct(10, "Hello anonymous structs")

ros_publisher_factory(topic, type, history="default", depth=1, reliability="default", durability="default", deadline=0, lifespan=0, liveliness="default", lease_duration=0)

Create a ROS2 publisher handle.

For more details regarding ROS2, consult ros.org. Each of the quality of service (QoS) parameters maps to the backend, and further details can also be found in the ROS2 documentation. https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html

Publisher and subscriber must have a compatible set of QoS parameters.

Parameters: (default will fall back on the ros backends default)

topic: The name of the topic.

type: The type of the messages on the topic. A dynamic library corresponding to the type must be present on the robot, e.g libstd_msgs__rosidl_typesupport_introspection_c.so for std_msgs/String.

history: How many messages to store in the history. Options are "default", "keep_last" and "keep_all".

depth: The size of the queue of old messages. It requires history to be "keep_last" to take effect.

reliability: How reliable the topic should be. Options are "default", "reliable", and "best_effort".

durability: Options are "default", "transient_local", and "volatile".

deadline: A contract for how much time is allowed between messages. **(not guaranteed to be part of first release)**

lifespan: The amount of time a message will be considered valid. 0 will mean lifespan is disabled. **(not guaranteed to be part of first release)** liveliness: Specifies for how long the publisher is considered alive. Options are "default", "automatic", and "manual". **(not guaranteed to be part of first release)**

lease_duration: The period of time a publisher has to indicate that it is alive before it is considered to have lost liveliness **(not guaranteed to be part of first release)**

Methods on the handle:

- write(myStruct) Takes one parameter: A struct that matches the message type of the topic.
- close() Indicates that the publisher will not be used further, and releases memory. Method should be used to actively close handles created in local scope. **(not implemented yet)**

Example command: ros_publisher_factory("test", "std_msgs/String")

ros_subscriber_factory(topic, type, history="default", depth=1, reliability="default", durability="default", deadline=0, lifespan=0, liveliness="default", lease_duration=0)

Create a ROS2 subscriber handle.

For more details regarding ROS2, consult ros.org. Each of the quality of service (QoS) parameters maps to the backend, and further details can also be found in the ROS2 documentation. https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html

Publisher and subscriber must have a compatible set of QoS parameters.

Parameters: (default will fall back on the ros backends default)

topic: The name of the topic.

type: The type of the messages on the topic. A dynamic library corresponding to the type must be present on the robot, e.g libstd_msgs__rosidl_typesupport_introspection_c.so for std_msgs/String.

history: How many messages to store in the history. Options are "default", "keep_last" and "keep_all".

depth: The size of the queue of old messages. It requires history to be "keep_last" to take effect.

reliability: How reliable the topic should be. Options are "default", "reliable", and "best_effort".

durability: Options are "default", "transient_local", and "volatile".

deadline: A contract for how much time is allowed between messages. **(not guaranteed to be part of first release)**

lifespan: The amount of time a message will be considered valid. 0 will mean lifespan is disabled. **(not guaranteed to be part of first release)**

liveliness: Specifies for how long the publisher is considered alive. Options are "default", "automatic", and "manual". **(not guaranteed to be part of first release)**

lease_duration: The period of time a publisher has to indicate that it is alive before it is considered to have lost liveliness **(not guaranteed to be part of first release)**

Methods on the handle:

- read() returns the last published message on the topic. It will block until one is available.
- wait(timeout) returns a boolean indicating when a message is available on the topic. It will block for at most "timeout" seconds.
- close() indicates that the subscriber is done, and releases memory. Method should be used to actively close handles created in local scope. **(not implemented yet)**

Example command: ros_subscriber_factory("test", "std_msgs/String")