Structs (complex data types) in URScript

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.
    a. Using the struct function
    b. Using the make_anon_struct function
    c. Doing an RPC call that returns a struct
    d. 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.

```
# 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.

```
# 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.

New script 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")

Disable/enable High Holding Torque

Currently the UR controller automatically enables unlimited torque window (high holding torque) when

- the program state is PROGRAM_STATE_RUNNING
- actual joint movement <= 0.01 rad/s
- target joint velocity == 0

It is useful to prevent a false positive protective stop when the robot is standing still while the end-effector is doing some work such as screw driving. However, if the robot is being transported on an external axis, then high torque hold should be disabled. This will allow collision detection of the stationary robot while being transported.

New URScript functions:

- high_holding_torque_disable()
- high_holding_torque_enable()

can be called to manually disable/enable high holding torque. When disabled, the robot should report protective stop if it collides with an obstacle when standing still. Note that the setting is reset to **enabled** after restarting the controller.

Modbus client improvements

Improvements are focused on accessing multiple coils or registers within single modbus transaction. This allows increasing overall communication efficiency when accessing devices like I/O expanders, and enables complex devices like servo drives. Up to 123 coils or registers can be read or written in single modbus transaction.

In addition new Modbus Function Codes are supported enabling broader range of devices compatible with UR robots.

New error handling functions added for more flexible, and error tolerant programs.

NOTE: Current release introduces new features in script only. Modbus signals added in installation still are limited to 1 coil/register. Same limitation applies to current value preview in I/O tab. Error handling/watchdog functions can be applied to signals both added in installation, and script.

New function codes support

Existing implementation of Modbus Client was limited to 1 coil, and 1 register per signal in the installation. Moreover only Function Codes (FC) 1-6 were available.

Now additional Function Codes 15, 16, and 23 are supported.

Matrix of function codes used for accessing coils or discrete inputs:

	Read		Write	
Signal type	**Single Coil**	**Multiple Coils**	**Single Coil**	**Multiple Coils**
0 = "Digital Input"	2	2	-	-
1 = "Digital Output"	1	1	5	15
15	1	1	15	15

Matrix of function codes used for accessing Input Registers, or Holding Registers:

	Read		Write	
Signal type	**Single Register**	**Multiple Registers**	**Single Register**	**Multiple Registers**
2 = "Register Input"	4	4	-	-
3 = "Register Output"	3	3	6	16
16	3	3	16	16
23	23	23	23	23

Signal types 0, 1, 2, 3 have extended functionality, but when one coil or register is specified, then different function code is used for writing. This behavior ensures backwards compatibility with software 5.14 and earlier.

Types 15 and 16 use the same signal type regardless of number of registers supplied.

Type 23 uses special modbus_rw_signal_add(...) function and allows to read and write multiple registers in single modbus transaction.

Improved error handling

In previous software versions accessing signal that either can't reach remote device, or device is reachable but reports Modbus exception would pause the program with "Protective Stop".

There are introduced few script functions for reading current error, and configuring watchdog time.

Script functions

Short description of each added, or modified script functions:

- `modbus_add_signal(IP, slave_number, signal_address,signal_type, signal_name, sequential_mode=False, register_count=1)` - introduced register count. For backwards compatibility, default value is 1.
- `modbus_add_rw_signal(IP, slave_number, read_address, read_register_count, write_address, write_register_count, signal_name, sequential_mode=False)` - new script function for adding signals for FC23. Read and write address spaces can be in different locations, and have different register count.
- `modbus_get_signal_status(signal_name, is_secondary_program)` - returns array of values when used on signal with multiple registers
- `modbus_set_output_register(signal_name, register_value_array, is_secondary_program)` - accepts array of values when used on signal with multiple registers
- `modbus_set_output_signal(signal_name, register_value_array, is_secondary_program)` - accepts array of values when used on signal with multiple coils
- modbus_get_error(signal_name, error_source=0) - returns error status of signal.
- modbus_set_signal_watchdog(signal_name, timeout_sec) - set tolerance for errors (both communication, and device exceptions) as minimum time between valid device responses.

- modbus_get_time_since_signal_invalid(signal_name) - returns time elapsed since signal entered error state.
- `modbus_reset_connection(connection_id, is_blocking=True)`- tear down, and reconnect all signals to remote device. Useful in error recovery routines.

Examples

Examples programs and script code provided for controlling Festo CPX Modbus Modular Terminal, and CMMT-AS servo drive.

Modbus server improvements

Added General Purpose bit registers to coils map. Registers are shared with RTDE boolean registers.

From robot program side registers can be accessed with script functions:

- read_input_boolean_register(address)
- read_output_boolean_register(address)
- write_output_boolean_register(address, value)

COIL ADDRESS			
Address	**Read**	**Write**	**Description**
300-428	✓	✓	128 General Purpose Input boolean registers (shared with RTDE boolean registers)
500-628	✓	✗	128 General Purpose Output boolean registers (shared with RTDE boolean registers)

Kinematic Tree in URScript

As of Polyscope 6, URScript programmers will be able to interact with coordinate frames as objects. The concept of coordinate frames isn't new and would be well known to any URScript programmer with a background in robotics. Much of the new functionality that we've implemented could have been accomplished by a knowledgeable roboticist along with use of *pose_trans*, *pose_inv*, etc. We've tried to make it easier for less knowledgeable programmers to accomplish the things that more experienced roboticists can do. As for the more experienced roboticists, we've tried to make it so their URScript programs can be simpler, easier to read, and more maintainable (reduce the use of *pose_trans*, *pose_inv,* etc.)

Here's a high-level overview of what's new:
Please see separate documentation package for this extensive feature.

Coordinate Frames as Objects

Added the concept of a coordinate frame as an object in the URScript language. Coordinate frames can be:

- given unique names like "frame1" or "table corner"
- referred to by their name in many existing motion commands (e.g., *movel, movep, movec,* etc)
- added and removed while a program is running
- attached and detached from one another
- moved around
- assigned inertial properties (e.g., mass and inertia)

Programs Don't Have to be Robot-centric

There are three "special" frames that always exist in a URScript program

- **"world"** is the only truly stationary frame. It cannot be moved or attached to any other frame.
- **"base"** corresponds to the robot's base frame. It is always attached to the "world" frame and can be moved around.
- **"tcp"** corresponds to the tool-center point and matches up with the existing TCP concept in URScript. It is placed relative to the flange using the existing *set_tcp* command

Traditionally, URScript programs have been mostly robot centric, meaning that all the motion commands (movel, etc.) take goal poses expressed in the robot frame. The consequence of this is that if the robot's base moves in space, the goal poses being passed to the motion commands must be updated to account for the robot's base motion. Most of the builtin URScript commands have been expanded in capability to accept 1) the name of a coordinate frame as a goal, and 2) a pose expressed in a specified frame as a goal. For example, if a program has a frame called "frame1", a call to movel("frame1") would command the robot to move the tcp's pose to match the pose of "frame1" in the world frame. The controller does the math so that the programmer does not need to always specify goal poses to the motion commands in the robot frame.

Backwards Compatibility: only added new capabilities to the motion commands. All existing ways of using the motion commands work fully and are unchanged so URScript programs that are working now without using these new features will continue to work just as they always have.

Manipulating the Kinematic Tree

Being able to attach and detach coordinate frames lets programmers create trees of attached frames. We call these kinematic trees. To be as precise as possible, the controller only knows one kinematic tree with the "world" frame at its root. When programmers attach frames to each other, they aren't creating new kinematic trees, they are manipulating sub-trees of the controller's kinematic tree. Here are some basic properties:

- The "world" frame is the root of the kinematic tree.
- The "tcp" is always attached to the robot "base" frame so that the tcp moves both if the robot's joints move but also if the robot's "base" frames moves.
- The robot "base" frame is always attached to the "world" frame but it can be moved.
- Moving a frame also moves its attached children frames in the same way.
- New frames that a programmer adds are attached to the "world" frame by default until they are attached elsewhere.
- New frames can be attached to any existing frame including the "tcp" and the robot's "base" frame.
- Frames that are attached to the "tcp" frame also move when the robot's joints move.

Payload Computation

Frames can be assigned inertial properties which can be used to compute the robot's payload when they are attached to the robot's TCP frame. This could be useful for pick-and-place tasks when the object inertial properties are known. In this case, the inertial properties of the pickup tool (gripper or suction cup) could be assigned to the "tcp" frame and a frame could be placed that represents the part being picked also with inertial properties assigned. When the part frame is attached to the "tcp" frame, the inertial properties of the combined masses and inertias can be automatically computed and set as the total robot payload.

Backwards Compatibility: we have only added new capabilities and existing URScript payload commands remain unchanged.

Motion in a Moving Frame

We created a new feature called "frame tracking". It's built on top of the existing "path offset" feature and lets users specify the world model frame that a robot trajectory should be performed in. The "frame tracker" updates the robot's motion to follow the trajectory in the tracked frame even while the frame moves in time, provided the frame's motion is smooth and continuous. To use this feature, the programmer adds a

frame to the kinematic tree, enables frame tracking on the frame, and then moves the frame with the move_frame() command.

Here are some interesting facts:

- Frame tracking is a key part of the coordinated external axis project (see here) for welding, where the robot is commanded to follow a weld trajectory defined in the moving frame of an external axis positioner.
- The frame tracker naturally handles motion of the robot base without any additional input from the user. This would be useful if the robot is mounted on a rail.

Change log

INU	3/28/2023