Matlab + RTI DDS

RTI DDS

This example is a **guideline** on how to write a Matlab script that, together with the Middleware, communicates with a vehicle. This tutorial is based on our platoon example (which is just a series of vehicles following a preset path without much logic behind that behavior), which you can find in

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/high_level_controller/examples/matlab/platoon

More specifically, we well take a look at `main_vehicle_ids.m`.

As you can see in the following sections, this script uses another script, called `init_script.m.`

> **Init script**
>
> The **init script** is similar (in its idea) to the cpm library for your C++ programs: It is supposed to **take care of redundant work** for you. In this case, it sets up all relevant **DDS participants** for the communication within the network. It also loads the required **IDL files** (message types), loads **XML files** for QoS settings and sets the variable for the **stop signal**, which will be mentioned later on. It currently requires a given folder structure, which is automatically set up on the NUCs and should also be part of the software repository's structure - in an ideal case, you do not need to worry about that, as long as you import the init script from its original place (it can also be found in ~/dev/software/high_level_controller /examples/matlab/ on the NUCs).
>
> You can find it here:
>
> https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/matlab/init_script.m

> **Working with RTI DDS in Matlab**
>
> Please consult RTI DDS and Matlab to get an idea of how to work with DDS in Matlab.

Your Matlab script is **not supposed to directly communicate with the vehicle** or to perform timing operations with the LCC. This is the Middleware's task, so your task is **just to properly communicate with the Middleware** (if you only want to control the vehicle).

The Middleware will wake your script up regularly to perform a computation with the currently given values, and you just need to respond in a proper way. This allows you to **focus on problem solving instead of timing and communication**. Always ensure that the middleware's period in the LCC "Parameters" tab is set according to your expectations in the HLC.

Setup

The structure of your code

Your function head may differ, but you **must use varargin** as your last parameter to pass vehicle IDs to your script. This allows you to define, for your script, which vehicle(s) it should be responsible for. Any previous parameters you define are your own 'custom' parameters and **need to be specified as additional parameters** in the Lab Control Centers' UI before starting your script (if you wish to start it using the UI).

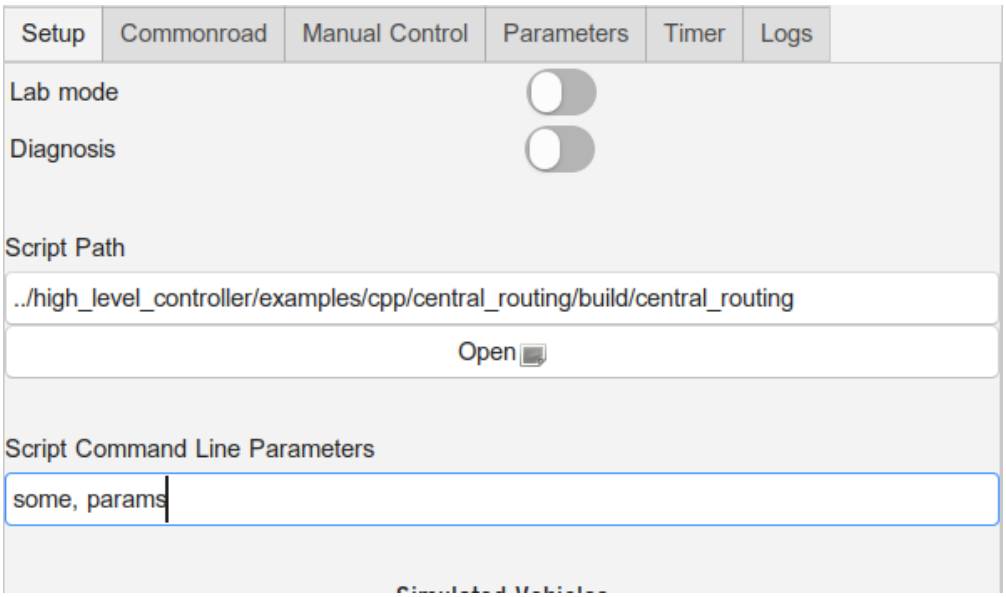
```
function main(middleware_domain_id, varargin)
...
vehicle_ids = varargin;
```

> **The parameter middleware_domain_id only exists in the eprosima branch and will not be added to the RTI Branch anymore! Don't use this parameter if you use RTI DDS.** Furthermore, the examples below use the RTI DDS syntax. eProsima syntax can be found here.

In other words: If your function head looks like this

```
function main(some, params, middleware_domain_id, varargin)
```

then you need to pass `some, params` as additional parameters in the LCC's UI (ignore the script path in the screenshot):



Varargin covers the vehicle IDs that your script should be responsible for. The DDS Domain ID to communicate with the Middleware only via shared memory is middleware_domain_id, its default value is 1. To be compatible with the LCC, which expects these two parameters to be available on calling your script, you should always put them as the last two parameters in your function head as specified above.

What your HLC scripts need to include

Import the init script

You can see the sample code below. What you should do:

1. Set the Matlab domain ID to 1 in a variable of your choice. This domain is used to communicate directly with the Middleware. More information can be found here.
2. Go to the right directory for the init script
3. Call the script and store the readers and writers that it sets up for you, as well as the stop signal (Which we call trigger_stop here).
4. Store the vehicle IDs from varargin. You will need them later.
5. The Middleware uses VehicleStateList data for timing purposes, i.e. if new data is received, you need to use it for your calculation, otherwise you should wait until you receive more data. This is further explained down below. A waitset is set to wait upon taking data from the reader if no data is yet available. The timeout is set to 10 seconds in this example, but you can also set a higher timeout.

```
function main_vehicle_ids(middleware_domain_id, varargin) % middleware_domain_id only for eProsima + Matlab
        % Set the matlab domain ID for communicating with the middleware (which is always 1)
    matlabDomainID = 1;

        % Clear command window, store path of current script, go to that path
    clc
    script_directoy = fileparts([mfilename('fullpath') '.m']);
    cd(script_directoy)

    % Set the path of the init script relative to the path of your script (or absolute), assert that it exists,
add it to the path so that it can be found and than call it
        % Then go back to the script directory
    init_script_path = fullfile('../', '/init_script.m');
    assert(isfile(init_script_path), 'Missing file "%s".', init_script_path);
    addpath(fileparts(init_script_path));
    [matlabParticipant, stateReader, trajectoryWriter, systemTriggerReader, readyStatusWriter, trigger_stop] =
init_script(matlabDomainID);
    cd(script_directoy)

        % Remember the set vehicle IDs in a new variable
    vehicle_ids = varargin;

    %% Use a waitset for the reader of vehicle states, which is later used to indicate if a new computation
should be started
        % The waitset makes sure that take() or read() from the reader's storage does not return if no new data
is available (until the timeout is reached, here 10 seconds
        % so that one can still check for a stop signal regularly)
    stateReader.WaitSet = true;
    stateReader.WaitSetTimeout = 10;

    %% Do not display figures
    set(0,'DefaultFigureVisible','off');
```

Tell the Middleware that your script is ready to operate

You must send a ReadySignal message **after initialization** - only the **ID string matters**, which must be of the form "`hlc_" + vehicle_id`, where the latter is the ID of the vehicle the HLC and thus your script is responsible for:

```
% (Assuming that vehicle_ids contains the IDs your script is responsible for)
% Send first ready signal to the Middleware to indicate that your program is ready to operate
% The signal needs to be sent for all assigned vehicle ids separately (usually, one script on one NUC only
manages one of the IDs)
% Also works for simulated time - period etc are set in Middleware, so you can ignore the timestamp field
for i = 1 : length(vehicle_ids)
        % Create a new msg of type ReadyStatus (idl)
    ready_msg = ReadyStatus;

        % Set the values of ready_msg
    ready_msg.source_id = strcat('hlc_', num2str(vehicle_ids{i}));
    ready_stamp = TimeStamp;
    ready_stamp.nanoseconds = uint64(0);
    ready_msg.next_start_stamp = ready_stamp;

        % Send the ready msg
    readyStatusWriter.write(ready_msg);
end
```

You use the `ReadyStatus` type here, which was defined in one of the IDL files that was imported for you using the init script. You can see the values of this type in the IDL file:

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/dds_idl/ReadyStatus.idl

This message **tells the Middleware that your script is now ready to operate** and to receive its data. Thus, at this point, your reader for vehicle states (here `stateReader`) should be initialized so that it can receive the data.

Consider the timing signals

This is very important. There are two signals you must consider: Start and stop signals. You have to **check regularly for the stop signal**, and **once for the start signal**:

- Start signal: This indicates that your script is supposed to calculate a new e.g. trajectory for the vehicles it is responsible for. Start with calculation immediately and send the results back as soon as possible. The Middleware handles the rest.
- Stop signal: `trigger_stop = uint64(18446744073709551615)` was already set by the init script. It is the highest number representable with a uint64_t type. This number indicates that the simulation was stopped, and thus your script should immediately terminate.
- All other numbers are irrelevant. From here on, **new messages received by the reader of `VehicleStateList (stateReader)`, are interpreted as start signals for less redundancy.**

Waiting for the initial start signal may look like this:

```
% Wait for start signal / stop if a stop signal was received
got_stop = false;
got_start = false;
while(true)
        % Read the newest system trigger message, if one exists
    trigger = SystemTrigger;
    sampleCount = 0;
    [trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);

        % Go through all received messages since the last check, if they exist (sampleCount gives the number of
read messages)
        % Any received message would indicate "start", but you have to check if a stop signal was received as
well
    while sampleCount > 0
                % Check for stop signal
        if trigger.next_start().nanoseconds() == trigger_stop
            got_stop = true;
        elseif trigger.next_start().nanoseconds() >= 0
            got_start = true;
        end

                % Read the next sample and continue until sampleCount is zero again (all samples have been read
then)
        [trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);
    end

        % Stop the loop if a signal was received, then proceed with handling a stop signal (stop the program)
or a start signal (continue)
    if got_stop | got_start
        break;
    end
end
```

You need to repeat this procedure to check for the stop signal every time before you wait for the next start indicator sent by the Middleware (which is indicated by new samples in `VehicleStateList`). The loop, that includes calculating new data as well, may look like this:

```
% Again: Go through all samples and stop the program if you received a stop signal
% break_while is the name used in the sample program (as it breaks the enclosing while-loop of the whole
program), but you could also name it e.g. stop_received
trigger = SystemTrigger;
sampleCount = 0;
[trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);

break_while = false;

while sampleCount > 0
    current_time = trigger.next_start().nanoseconds();
    if current_time == trigger_stop
        break_while = true;
    end

    [trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);
end

if break_while
    ...
```

Important

The Middleware might already have been stopped by a stop signal while you are waiting for the next start signal in form of a VehicleStateList. It is thus important to set a timeout for stateReader that is not too high, so that you can check for stop signals in between waiting for a new start signal. On the other hand, you want to react as fast as possible to a new start signal, so the waiting time should also not be too low - only consider stop signals if you can be sure that you should have received a new start signal by that time, i.e. after 10 seconds.

Receive information about your vehicle

Information about your vehicle are contained in the `VehicleStateList` messages, which include the c**urrent states and observations of all vehicles** as well as the **current time**. This signal is supposed to be the start signal for the HLC, so computation should start using this data directly after the message was received.

These signals, which, in this example, you can obtain using the `stateReader`, contain two fields which are vital to the correct usage of your script.

1. The messages contain current information about the **whereabouts and settings of your vehicle as well as any other vehicle** in the simulation. Use this data **for planning**.
2. The messages contain **timing information**. Use these when you send commands to your vehicle - they contain the current time you set for the messages.

This signal has a third purpose as well. It shows the script when to start the computation, similar to a timing signal. The desired computation cycle is the following:

- **Wait** for a new message
- After receiving a message, **start your computation**
- Then **send** the new **command**(s) to the vehicle
- Remove messages that were received during computation, as they are old and potentially unusable, and **wait** for the next message
- ... (repeat)

The history for this signal is set to 1, but you may still get an outdated signal here **if you missed a period during** your **computation** and read the next VehicleStateList in the middle of that next period. In that case, it may be **better** to **skip** that **next period** as well and wait for the following one to start.

```
...
% Read the current vehicle information from the according reader, if such data is available
% Due to the set waitset, you wait for up to 10 seconds at take(...) if no data is available
sample = VehicleStateList;
status = 0;
stateSampleCount = 0;
sampleInfo = DDS.SampleInfo;
[sample, status, stateSampleCount, sampleInfo] = stateReader.take(sample);

% Check if any new message was received, then proceed with handling the data, computing your solution and
finally sending back a command to the vehicle(s) your script controls
if stateSampleCount > 0
    ...
```

Send commands to your vehicle

You need to send vehicle command messages as a result of your computation including the vehicle ID to the Middleware, which propagates these to the vehicle. The implementation of the computation of e.g. the vehicle's trajectory is not explained here and depends on your task. You are only given an example of how to send a simple trajectory here.

We are now taking a look at In the script which we were looking at `leader.m`, which, as you can see in the first code sample below, is called within the platoon script. We use it to compute the next trajectory segment:

Important

All the scripts that you use internally **should be in the same folder**, else they will not work when they are being deployed remotely.

```
% Call the programs which calculate the trajectories of all HLCs, then send the results back to the vehicles
if (size(vehicle_ids) > 0)
    for i = 1 : length(vehicle_ids)
        msg_leader = leader(vehicle_ids{i}, sample.t_now);

                % Send resulting trajectory to the vehicle
        trajectoryWriter.write(msg_leader);
    end
end
```

In the following, you see an example of how to set up a trajectory message, that can then be sent using the `trajectoryWriter`:

```
%Create msg of type trajectory
trajectory = VehicleCommandTrajectory;

% Set the vehicle ID
trajectory.vehicle_id = uint8(vehicle_id);

% In this example, we only send one trajectory point (which is not sufficient for interpolation)
trajectory_points = [];
point1 = TrajectoryPoint;

% This trajectory point should be considered in the future, so add some nanoseconds to the current time, then
set the time stamp for the trajectory point
% t_eval here is the time that we got from the middleware in the VehicleStateList message - you are not
supposed to use your own clock / timer (else you would have trouble working with simulated time)
time = t_eval + 400000000;
stamp = TimeStamp;
stamp.nanoseconds = uint64(time);
point1.t = stamp;

% Set other trajectory values - position and velocity in 2D coordinates
point1.px = trajectory_point(1);
point1.py = trajectory_point(2);
point1.vx = trajectory_point(3);
point1.vy = trajectory_point(4);

% Append to the list of all trajectory points
trajectory_points = [trajectory_points [point1]];
trajectory.trajectory_points = trajectory_points;

% Send the trajectory to the vehicle
trajectoryWriter.write(trajectory);
```

Further Information

Bash script / Deploy Using the LCC

If you want to start your script using a bash script, you can do it like this:

```
/opt/MATLAB/R2019a/bin/matlab -logfile matlab.log -sd $script_dir -batch "$script_name(1, ${vehicle_id})"
```

Usually, you would want to start it using the LCC instead, which takes care of these things for you.

Deploying with Matlab GUI

See here: If, within the setup tab, you do not select any script but leave the script field empty, only the Middleware gets started. You can then start your own script from within the Matlab GUI, it should detect the Middleware (if you set the correct number of vehicles, the Middleware will, if deployed locally, wait for HLC messages for all vehicles that are currently online).

How to Actually Start Your Script

As mentioned before, the Middleware takes care of timing. As soon as your script has told it that it is ready to operate, the Middleware should appear in the Timer Tab in the LCC. Then, you can start the simulation by sending a start signal (press on the start button). Of course, you need to deploy / start the simulation to start the Middleware before you start the timer (and, if you do not use the Matlab GUI, your selected Matlab script as well).