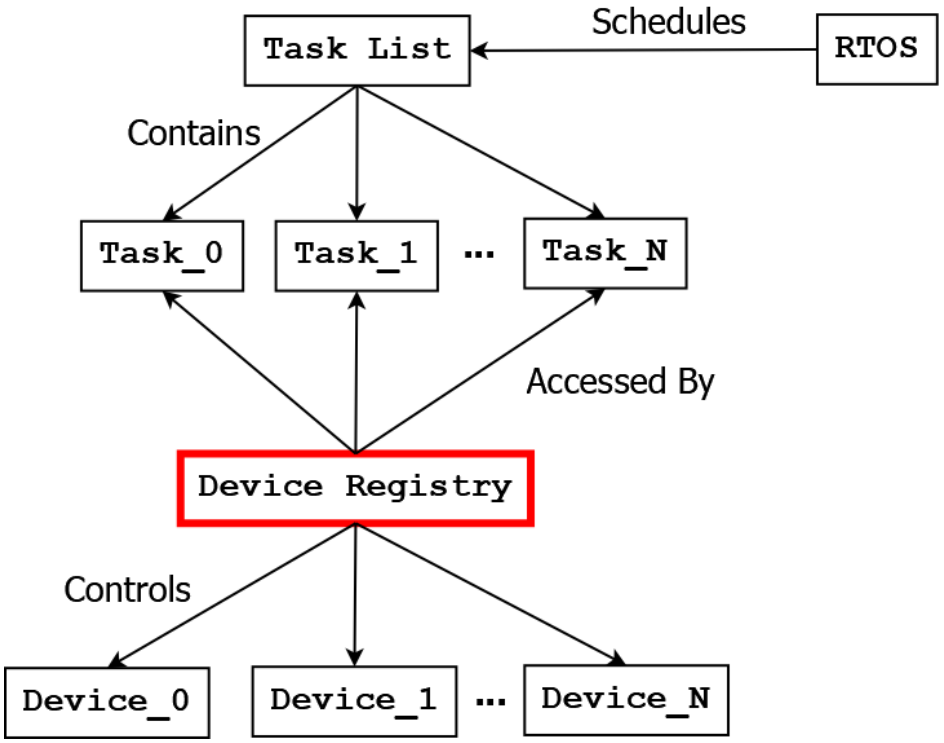
Device Registry



Abstract

The Device Registry class serves as the primary point of interaction between student-created Tasks and the underlying physical devices used by the Ego vehicle. A device can be either a sensor that perceives the environment, or an actuator that performs certain actions. An example of a sensor is a camera, and an example of an actuator is a braking system. Students are only allowed to control the Ego vehicle's devices through the simulated system busses which are exposed by the device registry.

This setup is analogous to real world RTOS based robotics, which often have numerous independent subsystems that interact through tightly controlled interfaces. While the AVL makes some simplifying assumptions on the architecture, the general ideas are still the same as those used in a real-world application.

Device Control

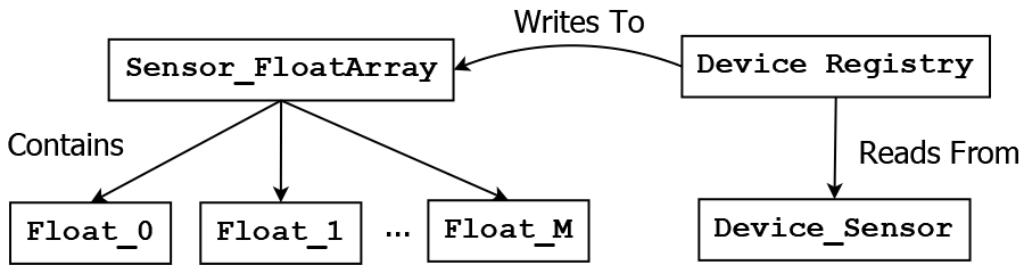
Each controllable device in the Ego vehicle has a simulated system bus within the Device Registry class. These simulated system busses take the form of arrays of floating-point numbers (aka floats). The sizes and shapes of these arrays can vary based on the devices they are associated with.

Both sensors and actuators have their own float arrays, however there are slight differences in how they are accessed/controlled. Furthermore, not all the sensors or actuators will be activated/usable in each assignment scenario. Read the documents for each assignment scenario to determine which devices are made available to student-created tasks. If a device is not made available, then its float array can still be accessed, however these float arrays will be ignored by the Device Registry during each frame update.

Device Control: Sensor

At the start of every frame, the Device Registry gathers the current readings from all of the sensors and places the readings the associated float array. Student-created tasks can then directly read these updated values from the float arrays which are contained within the Device Registry.

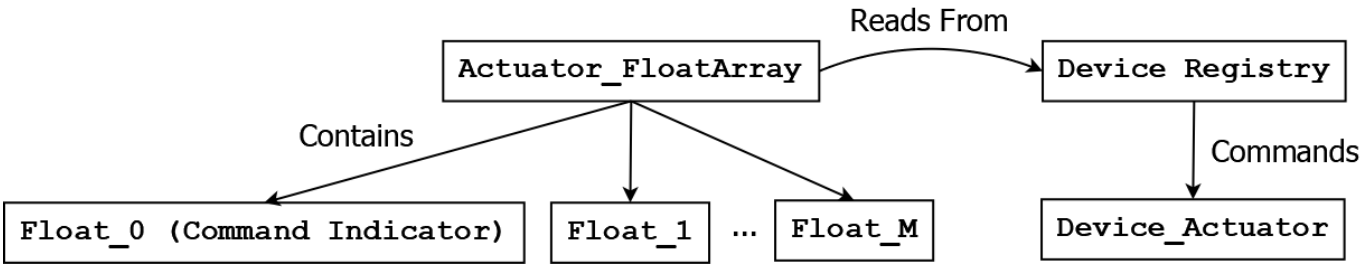
While it is possible for student-created tasks to write to a sensor's float array, this will have no effect on the sensor itself. Upon the start of the next frame, the Device Registry will overwrite this data with new sensor readings.



Device Control: Actuator

At the start of every frame, the Device Registry checks the first entry of each actuator's float array. The first entry of the array is the "command indicator", and if the value of the command indicator is anything other than 0 then the Device Registry will send a command to the actuator device. After the command is sent, the command indicator will be reset back to 0.

When a command is sent to an actuator, the remaining entries in the float array are sent to the actuator as control data. Each actuator device interprets the control data differently.



Device System Bus List

This section will describe the float arrays for each device's simulated system bus, as well as provide examples for how to access the float arrays in the code of a student-created task.

Sensor Devices

The system busses associated with each sensor can have different sizes and shapes. Particularly attention should be paid to the system busses associated with the lidar array and the color camera.

GPS Sensor : DeviceRegistry.gps

- **Description:** Provides the position of the robot in the world.
- **Shape:** [2]
- **Possible values:** (-infinity, +infinity) Exclusive
- **Contents of Float Array:**

Index	0	1
Contents	Latitude	Longitude

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float latitude = devices.gps[0];
        float longitude = devices.gps[1];

        // Print data to the console
        Debug.Log($"Latitude: {latitude}");
        Debug.Log($"Longitude: {longitude}");

    }
}
```

Lidar Array Sensor : DeviceRegistry.lidar

- **Description:** Provides the readings of the 16 lidar sensors that are placed on top of the Ego vehicle. When an object intersects the beam which is projected from a lidar sensor, the distance to that object is stored in an associated memory cell. When no object is within range of a lidar sensor, the maximum range value (10 meters) is stored instead.

  The lidar sensors are placed in a circle on top of the Ego vehicle, thus giving a full 360° view

around the robot. The indices of the lidar sensors increase by 1 each time as they rotate clockwise, e.g. index 0 gives the forward facing sensor, index 1 gives the sensor which is rotated 22.5°, index 4 gives the sensor which is rotated 90°, index 8 gives the sensor which is rotated 180°, and index 15 gives the sensor which is rotated 337.5°.

- **Shape:** [16]
- **Possible values:** [0, 10] Inclusive
- **Contents of Float Array:**

Index	0	1	…	15
Contents	Lidar 0°	Lidar 22.5°	…	Lidar 337.5°

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float forward_lidar = devices.lidar[0];
        float right_lidar = devices.lidar[4];
        float back_lidar = devices.lidar[8];

        // Print data to the console
        Debug.Log($"Forward Lidar: {forward_lidar}");
        Debug.Log($"Right Lidar: {right_lidar}");
        Debug.Log($"Back Lidar: {back_lidar}");

    }
}
```

Color Camera : DeviceRegistry.pixels

- **Description:** Provides the readings of the low-resolution color camera that is mounted on the Ego vehicle. The camera is 7 pixels high by 15 pixels wide, and each pixel contains 3 values that correspond to the RGB (Red-Green-Blue) values detected by the camera.
- **Shape:** [7, 15, 3]
- **Possible values:** [0, 255] Inclusive
- **Contents of Float Array:**

	Index	Contents			
Index		0	1	…	15
Contents	0	(R,G,B)	(R,G,B)	…	(R,G,B)
	1	(R,G,B)	(R,G,B)	…	(R,G,B)
	…	…	…	…	…
	7	(R,G,B)	(R,G,B)	…	(R,G,B)

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float top_left_red = devices.pixels[0, 0, 0];
        float middle_left_green = devices.pixels[7, 0, 1];
        float bottom_right_blue = devices.pixels[7, 15, 2];

        // Print data to the console
        Debug.Log($"Top left pixel: {top_left_red}");
        Debug.Log($"Middle left pixel: {middle_left_green}");
        Debug.Log($"Bottom right pixel: {bottom_right_blue}");

    }
}
```

Compass : DeviceRegistry.compass

- **Description:** Provides the difference in angle between the front of the Ego vehicle and the direction that faces North. If the robot is facing North, the value is 0. If the robot is facing West, then the value will be -90. If the robot is facing East, the value will be 90. If the robot is facing South, then the value is 180.
- **Shape:** [1]
- **Possible values:** (-180, 180] Exclusive lower, inclusive upper
- **Contents of Float Array:**

Index	0
Contents	Angle

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float angle = devices.compass[0];

        // Print data to the console
        Debug.Log($"Angle to North: {angle}");
    }
}
```

Target Compass : DeviceRegistry.targetAlignment

- **Description:** Provides the difference in angle between the front of the Ego vehicle and the direction to a specific point in the world. If the robot is directly facing the target position, the value is 0. If the robot is facing to the left at a perpendicular angle, then the value will be -90. If the robot is facing right at a perpendicular angle, the value will be 90. If the robot is facing the opposite direction of the target, then the value is 180.
- **Shape:** [1]
- **Possible values:** (-180, 180] Exclusive lower, inclusive upper
- **Contents of Float Array:**

Index	0
Contents	Angle

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float angle = devices.targetAlignment[0];

        // Print data to the console
        Debug.Log($"Angle to North: {angle}");
    }
}
```

Microphone : DeviceRegistry.microphone

- **Description:** Provides the frequency of a sound detected by the microphone that is mounted at the front of the Ego vehicle. If there is no sound in the environment, or if the Ego vehicle is too far away from a sound source, then the value 0 will be returned.

  The potential sound frequencies vary based on the specific assignment scenario. Review documentation for a specific scenario to find out what sound frequencies may be detected, and under what circumstances they will be heard.
- **Shape:** [1]
- **Possible values:** [0, infinity) Inclusive lower, exclusive upper
- **Contents of Float Array:**

Index	0
Contents	Sound Frequency

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float sound = devices.microphone[0];

        // Print data to the console
        Debug.Log($"Sound: {sound}");
    }
}
```

Speedometer : DeviceRegistry.speedometer

- **Description:** Provides the current speed of the Ego vehicle. The speed is given in meters-per-second. The ego vehicle has a top speed of 0.33 m/s.
- **Shape:** [1]
- **Possible values:** [0, 0.33] Inclusive
- **Contents of Float Array:**

Index	0
Contents	Speed

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Read data from the float array
        float speed = devices.speedometer[0];

        // Print data to the console
        Debug.Log($"Speed: {speed}");
    }
}
```

Actuator Devices

At the present moment, all actuator float arrays only have two entries. The first entry is the previously mentioned command indicator, which signals to the associated actuator device that a command is ready to be sent. The second entry contains command data which is used to control the device. All actuators interpret their command data in different ways.

When an actuator receives a command, the command indicator entry is reset back to 0. You can also read the current value

Speed Controller : DeviceRegistry.speedControl

- **Description:** Controls the movement speed of the Ego vehicle. When the command indicator is set to a non-zero value, the speed controller takes whatever data is stored at index 1 and uses it as the target speed. The speed controller will then accelerate the Ego vehicle towards this target speed, and it will maintain the Ego vehicle at that speed until a new command is given. The target speed is understood to be in meters-per-second.

  If the target speed is set to be less than the current speed, then the speed controller will not add acceleration nor will it apply brakes. Brakes will only be applied when the braking controller is explicitly commanded to apply them by a student-created task.

  If the target speed is set to a negative value, then the Ego vehicle will move backwards.

  Please note that the Ego vehicle has a max speed of 0.33 m/s. If you set the target speed to be above this maximum speed, then the Ego vehicle will only accelerate up to the max and no further.
- **Shape:** [2]
- **Possible values:** (-infinity, infinity) Exclusive
- **Contents of Float Array:**

Index	0	1
Contents	Command Indicator	Speed

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Set the command indicator
        devices.speedControl[0] = 1f;

       // Set the target speed to 0.2 m/s
        devices.speedControl[1] = 0.2f;
    }
}
```

Brake Controller : DeviceRegistry.brakeControl

- **Description:** Controls the brakes of the Ego vehicle. When the command indicator is set to a non-zero value, the brake controller takes whatever data is stored at index 1 and uses it as the brake time. The brake controller will then apply the brakes of the Ego vehicle for the amount of time that is given. The brake time is understood to be in seconds. Once this time expires, the brakes will no longer be applied.

  If the brakes are active and the brake time is set to 0 by a new command, then the brakes will no longer be active. If the brake time is set to a negative number, then this is treated as though the brake time is set to 0.

  Please note that the brake controller does not override the speed controller. If the brakes are active and the target speed of the speed controller is not 0, then the speed controller will win out and the vehicle will continue to move. Thus if you wish to come to a stop, then you must activate the brakes and simultaneously set the target speed to 0.

  You can also use the brakes to slow down to a new target speed very quickly. You can set the target speed to the new, slower value, and then activate the brakes. The speed controller only applies acceleration when the current speed is less than the target speed, so the brakes will slow down the vehicle until the new target speed is reached. Once the new target speed is reached, the speed controller will apply acceleration again.
- **Shape:** [2]
- **Possible values:** (-infinity, infinity) Exclusive
- **Contents of Float Array:**

Index	0	1
Contents	Command Indicator	Brake Time

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Set the speed controller to have a target speed of 0
        devices.speedControl[0] = 1f;
        devices.speedControl[1] = 0f;


       // Apply the brakes for 5 seconds
        devices.brakeControl[0] = 1f;
        devices.brakeControl[1] = 5f;
    }
}
```

Steering Controller : DeviceRegistry.steeringControl

- **Description:** Steers the Ego vehicle. When the command indicator is set to a non-zero value, the steering controller takes whatever data is stored at index 1 and uses it as the steering angle. This steering angle is then used by the steering controller to modify the direction the robot is facing. This modification of the direction occurs immediately at the start of the next frame, and it is relative to the current direction the Ego vehicle is facing. The steering angle is understood to be in terms of degrees, where a value 0 and 360 are considered equivalent.

  Positive values cause the Ego vehicle to turn right, and negative values cause the Ego vehicle to turn left. If an angle greater than 360 or less than -360 is given, then the angle will wrap back around. Thus steering angles of 10, -350, and 370 are all equivalent, and will result in the same final direction.

  If the Ego vehicle is facing North, and a steering angle of 90 is given, then the Ego vehicle will turn to face East. If the Ego vehicle is facing East, and a steering value of 90 is given, then the Ego vehicle will turn to face South. If the Ego vehicle is facing West, and a steering value of -90 is given, then the Ego vehicle will turn to face South.

- **Shape:** [2]
- **Possible values:** (-infinity, infinity) Exclusive
- **Contents of Float Array:**

Index	0	1
Contents	Command Indicator	Steering Angle

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Set the command indicator
        devices.speedControl[0] = 1f;

       // Set the steering angle to turn 30 degrees to the right
        devices.speedControl[1] = 30f;
    }
}
```

Transmitter : DeviceRegistry.transmitterControl

- **Description:** Transmits signals from Ego vehicle to receiver devices that are located within the environment scenario. When the command indicator is set to a non-zero value, the transmitter takes whatever data is stored at index 1 and uses it as the signal frequency to broadcast an “on” signal over. When this “on” signal is sent at the given frequency, any receiver devices which listen at the frequency will perform an action. The frequency is understood to be in mega-hertz (MHz).

  The potential receiver devices and their associated frequencies can vary based on the specific assignment scenario. Review documentation for a specific scenario to find out what frequencies may be received, and what actions the receiver devices will perform.

- **Shape:** [2]
- **Possible values:** (-infinity, infinity) Exclusive
- **Contents of Float Array:**

Index	0	1
Contents	Command Indicator	Frequency

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Set the command indicator
        devices.transmitterControl[0] = 1f;

       // Set the transmission frequency to 55.3 MHz
        devices.transmitterControl [1] = 55.3f;
    }
}
```

Miscellaneous Devices

In addition to traditional sensors and actuators, there may be other devices which can be used by student-created tasks.

Memory : DeviceRegistry.memory

- **Description:** A dedicated float array that student-created tasks can use to store and retrieve information. Some possible uses include keeping track of a system state, or allowing communication of data between different tasks. The memory float array will only be modified by student-created tasks. Thus if a value is placed in the array, it is guaranteed to stay there until another student-created task modifies it.

- **Shape:** [54]
- **Possible values:** (-infinity, infinity) Exclusive
- **Contents of Float Array:**

Index	0	1	…	63
Contents	Float_0	Float_1	…	Float_63

- **Example:**

```
public class Example_Task : TaskInterface
{
    public void Execute(DeviceRegistry devices) {
        // Store an arbitrary value in memory location 23
        devices.memory[23] = 69.420f;

        // Read a value from memory location 7
        float value = devices.memory[7];
    }
}
```