

```

/**
 * @file control-comms.hpp
 * @brief Communication interface for actions and observations over serial
 * @author Shawn Hymel
 * @date 2023-08-05
 * @version 0.1
 *
 * @details
 * This interface provides two main functions: receive_action() and
 * send_observation(). These functions are intended to provide an interface
 * between a controller or AI and an Arduino (i.e. for interacting with the
 * physical world).
 *
 * Your main Arduino program should first initialize a Serial object and pass
 * it into the init() function. In loop(), call receive_action(), which will
 * wait the set Serial timeout (default is 1 second) for a JSON string. If a
 * valid JSON string is received, the caller should perform the actions given
 * (user-defined). The caller should then return with an observation by calling
 * send_observation(). Just like with the action space, the observation space
 * is user-defined.
 *
 * NOTE: This interface requires the ArduinoJson v6 library. It was tested with
 * v6.21.3.
 *
 * Example:
 * @code{.cpp}
 * #include "control-comms.hpp"
 *
 * static constexpr size_t NUM_ACTIONS = 2;
 * static constexpr size_t NUM_OBS = 3;
 * ControlComms ctrl;
 *
 * void setup() {
 *     Serial.begin(115200);
 *     ctrl.init(Serial);
 * }
 *
 * void loop() {
 *     int command;
 *     float action[NUM_ACTIONS];
 *     float observation[NUM_OBS] = {3, 4, 5};
 *     ControlComms::StatusCode rx_code;
 *
 *     rx_code = ctrl.receive_action<NUM_ACTIONS>(&command, action);
 *     if (rx_code == ControlComms::OK) {

```

```

*     ctrl.send_observation(0, millis(), false, observation, NUM_OBS);
*   } else if (rx_code == ControlComms::ERROR) {
*     Serial.println("Error receiving actions");
*   }
* }
*
* @endcode
*
* @todo Create simple JSON parser to remove need for ArduinoJSON library
* @todo Rename RX keys to `rx_key_<item>`
* @todo Turn TX keys into `tx_key_<item>` member constants
*
* @copyright
* Zero-Clause BSD
*
* Permission to use, copy, modify, and/or distribute this software for
* any purpose with or without fee is hereby granted.
*
* THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
* WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE
* FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY
* DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
* AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
* OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

#include <Arduino.h>
#include <ArduinoJson.h>

/**
 * @brief Interface class to construct JSON objects to send and receive.
 */
class ControlComms {
public:

    /**
     * @brief Level of debugging info sent over serial.
     */
    typedef enum {
        DEBUG_NONE = 0,
        DEBUG_ERROR,
        DEBUG_WARN,
        DEBUG_INFO
    } DebugLevel;

```

```

/**
 * @brief Status codes returned by functions.
 */
typedef enum {
    OK = 0,
    RX_EMPTY,
    ERROR
} StatusCode;

/**
 * @brief Default constructor placeholder
 */
ControlComms() {

}

/**
 * Initialize serial object
 */
int init(
    Stream &serial_port,
    DebugLevel debug_level = DEBUG_NONE
) {

    // Assign our serial object
    stream = &serial_port;

    // Assign debugging
    debug = debug_level;

    return OK;
}

/**
 * @brief Send a JSON object with the observation out over the serial port.
 *
 * @param[in] status User-defined status code to send to computer
 * @param[in] timestamp User-defined timestamp (unsigned long integer)
 * @param[in] terminated True if the episode is over, false otherwise
 * @param[in] observation Array of values (floats) to send to the computer
 * @param[in] num_obs Number of values in the observation array
 * @param[in] digits Truncate the floats in the observation to this number
 */
void send_observation(
    int status,
    unsigned long timestamp,

```

```

    bool terminated,
    float *observation,
    size_t num_obs,
    uint8_t digits = 2
) {

    stream->print(F("{\"status\":}"));
    stream->print(status);
    stream->print(F(",\"timestamp\":}"));
    stream->print(timestamp);
    stream->print(F(",\"terminated\":}"));
    stream->print(terminated ? F("true") : F("false"));
    stream->print(F(",\"observation\":[}"));
    for (unsigned int i = 0; i < num_obs; i++) {
        stream->print(observation[i], digits);
        if (i < num_obs - 1) {
            stream->print(F(","));
        }
    }
    stream->print(F("]}"));
    stream->println();
}

/**
 * @brief Receive action command, parse actions into floating point array.
 *
 * @tparam num_actions Number of actions to be received
 * @param[out] command User-defined command to be received
 * @param[out] action_out Received actions are stored here
 *
 * @return OK, RX_EMPTY, or ERROR depending on the received string
 */
template <size_t num_actions>
StatusCode receive_action(int *command, float *action_out) {

    DeserializationError err = DeserializationError::Ok;
    StatusCode retcode = OK;

    // Figure out size of receive doc capacity
    constexpr unsigned int rx_doc_capacity =
        JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(num_actions + 1);
    StaticJsonDocument<rx_doc_capacity> doc;

    // Return early if nothing in receive buffer
    if (stream->available() <= 0) {

```

```

    return RX_EMPTY;
}

// Attempt to parse JSON
err = deserializeJson(doc, *stream);
if (err.code() == DeserializationError::Ok) {

    // Make sure the keys are present
    if (doc.containsKey(rx_action_key) &&
        doc.containsKey(rx_command_key)) {

        // Save command to output variable
        *command = doc[rx_command_key];

        // Save values to output array
        JsonArray vals = doc[rx_action_key];
        if (vals.size() == num_actions) {
            for (int i = 0; i < vals.size(); i++) {
                action_out[i] = vals[i];
            }

            // Array size mismatch
        } else {
            retcode = ERROR;
            if (debug >= DEBUG_ERROR) {
                stream->print("JSON Error: expected ");
                stream->print(num_actions);
                stream->print(" actions, got ");
                stream->println(vals.size());
            }
        }
    }

    // Key not found in doc
} else {
    retcode = ERROR;
    if (debug >= DEBUG_ERROR) {
        stream->print("JSON Error: keys ");
        stream->print(rx_command_key);
        stream->print("' or '");
        stream->print(rx_action_key);
        stream->println("' not found");
    }
}
}

```

```

// We'll get some invalid straggle chars over Serial, so ignore them

```

```

    } else {
        switch (err.code()) {
            case DeserializationError::EmptyInput:
            case DeserializationError::IncompleteInput:
            case DeserializationError::InvalidInput:
                retcode = RX_EMPTY;
                break;
            default:
                retcode = ERROR;
                break;
        }
    }

    // Flush receive buffer to avoid reading any extra '\r' or '\n' next time
    while (isspace(stream->peek())) {
        stream->read();
    }

    // Debug JSON parsing
    if (debug >= DEBUG_ERROR) {
        if (err.code() != DeserializationError::Ok) {
            stream->print("JSON Error: ");
            stream->println(err.f_str());
        }
    }

    return retcode;
}

private :
    Stream *stream;
    size_t rx_doc_capacity = 0;
    DebugLevel debug = DEBUG_NONE;
    static constexpr char rx_action_key[] = "action";
    static constexpr char rx_command_key[] = "command";
};

```