Error Handling Report

Sagar Syal

August 30, 2023

Contents

Introduction

Background

Hardware

General

The Dynamixel XH540-W150-T/R servo actuators, developed by Robotis, are part of a specialized series engineered for high-torque and high-speed applications. With a size of 88.23 cm^3 and a gear ratio of 150:1, these actuators are configured to deliver robust torque output, making them particularly suitable for demanding operational scenarios.

Dynamixel Control Table

The Control Table is a structured data framework comprising various data fields that are employed either for storing device status or for device control. Users can receive the current state of the device by querying specific data fields within the Control Table using READ Instruction Packets. Conversely, WRITE Instruction Packets allow users to manipulate the device by modifying applicable data within the Control Table. Each data field in the Control Table is uniquely identifiable by its address. To execute read or write operations, users are required to specify this unique address within the corresponding Instruction Packet.

Area (EEPROM & RAM)

The Control Table is split into two distinct sections: the Electrically Erasable Programmable Read-Only Memory (EEPROM) and the Random Access Memory (RAM). The EEPROM section is non-volatile, ensuring that data values are preserved even after the device is powered down. In contrast, the RAM section is volatile; its data values are only maintained when the device is powered on and will revert to initial settings upon restart.

Size

The size of the data fields within the Control Table can range from 1 to 4 bytes, depending on their specific function and usage requirements.

Access

The Control Table features two distinct access properties: 'RW' and 'R.' The 'RW' property signifies that the data field permits both read and write access, whereas the 'R' property indicates read-only access. Data fields designated as read-only ('R') cannot be modified through WRITE Instructions and are generally employed for measurement and monitoring functions. Conversely, data fields with read-write ('RW') permissions are typically used for dynamic device control.

Initial Value

Upon device initialization, all data fields in the Control Table are reset to their initial values. For the EEPROM section, any user-modified values become the new initial values, while the RAM section consistently reverts to predetermined initial settings.

Important Addresses on the Table

EEPROM Area

- ID (7)
  - The ID serves as a unique identifier for distinguishing between multiple devices in a network, enabling targeted communication and control.

- Operating mode (11)
  - The Operating Mode defines the specific behaviour and functional characteristics of the dynamixel.
    - Current Control Mode:
      - This mode focuses on torque regulation and is highly applicable for systems that prioritize torque control, such as gripping mechanisms.
    - Velocity Control Mode:
      - Designed to control actuator speed, this mode is analogous to the Wheel Mode in other Dynamixel products. It is well-suited for mobile robotic platforms requiring speed control.
    - Position Control Mode:
      - Serving as the default mode, it aims to control the actuator's position within a pre-defined rotational range and is of particular interest for articulated robotic systems with limited rotational capability.
    - Extended Position Control Mode:
      - This mode allows for up to 512 rotations, making it suitable for systems such as multi-turn wrist joints or conveyor mechanisms.
    - Current-Based Position Control Mode:
      - Integrating features of both current and positional control, this mode allows for up to 512 rotations. It is intended for complex robotic applications requiring both torque and position control.
    - PWM Control Mode (Voltage Control Mode):
      - This mode enables direct control over the Pulse Width Modulation (PWM) output, providing an additional layer of control granularity.
- Max Voltage Limit (32)
  - To determine the maximum operating voltages.
- Min Voltage Limit (34)
  - To determine the minimum operating voltages.
- PWM Limit (36)
  - To determine the maximum PWM output.
- Current Limit (38)
  - To determine the maximum current output limit.
- Velocity Limit (44)
  - To determine the maximum goal velocity.
- Max Position Limit (48)
  - The Min Position Limit defines the lower boundary for the actuator's motion when it is operating in Position Control Mode.
- Min Position Limit (52)
  - The Max Position Limit defines the upper boundary for the actuator's motion when it is operating in Position Control Mode.

RAM AREA

- Torque Enable (64)
  - The Torque Enable controls the torque status; writing a '1' to this address not only activates the torque but also locks all modifiable fields in the EEPROM area.
- Hardware Error Status (70)

  - Signifies the presence of a hardware error. The torque is immediately set to zero and locked. Reactivating the torque requires either a REBOOT command or a full power cycle of the servo. Errors are not mutually exclusive.
  - Bit 5: Overload error
    - Indicates a persistent load that surpasses the device's maximum output capacity.
  - Bit 4: Electrical Shock Error
    - Indicates there is an electrical shock on the circuit or there is not enough power to operate the servo.
  - Bit 3: Motor Encoder Error
    - Indicates there is a malfunction in the motor encoder.
  - Bit 2: Overheating Error
    - Indicates the internal temperature exceeds the set limit.
  - Bit 0: Input Voltage Error
    - Indicates that the voltage exceeds the configured limit.
  - Bits 7,6 and 1 are unused.
- Goal PWM (100)
  - In PWM Mode, the value stored at the designated address directly modulates the motor's behaviour via an inverter. The value cannot exceed the PWM limit.
- Goal Current (102)
  - In current control mode and Current-based Position Control Mode, the value at this address establishes a current limit. The value cannot exceed the current limit.
- Goal Velocity (104)
  - In velocity control mode this register is used to set the velocity of the servo.

Protocol 2.0

Instruction packet

To interface with the architecture with Dynamixel XH540-W150-T/R actuators, a critical element is the deployment of Instruction Packets, which serves as the communication vessels. In the communication protocol for Dynamixel XH540-W150-T/R actuators, packets are constructed as sets of 13 bytes encapsulating key parameters vital for device control and data transmission. These bytes include but are not limited to, an instruction byte specifying the command to be executed, a device ID byte to identify the target actuator, and a length byte to indicate the size of the packet. The 13-byte packet format streamlines the exchange of information between the control system and the actuator, enabling efficient, reliable, and precise command execution.

Header

In the communication protocol for Dynamixel XH540-W150-T/R actuators, each packet begins with an 8-byte header that serves as a flag to indicate the beginning of a new data packet.

Reserved

In the packet structure for the Dynamixel XH540-W150-T/R actuators, the reserved section is designated as 0x00 and serves as the end of the 6-byte header.

Packet ID

In the packet structure, for the Dynamixel XH540-W150-T/R actuators, the id designates the recipient actuator that should process the incoming command. The allowable range for this ID spans from 0 to 252, represented in hexadecimal as 0x00 to 0xFC, offering a total of 253 unique IDs that can be assigned.

Length

In the communication protocol for Dynamixel XH540-W150-T/R actuators, the Length field in the packet plays a crucial role in detailing the size of the packet. This field is divided into low and high bytes to efficiently represent the length. The Length value specifically accounts for the size of the Instruction, Parameters, and CRC (Cyclic Redundancy Check) fields within the packet. The formula for calculating the Length is the sum of the number of Parameters plus 3.

Instruction

In the communication schema utilized with Dynamixel XH540-W150-T/R actuators, a dedicated field exists within the Instruction Packet for specifying the type of command to be executed. This field serves as the instruction identifier, dictating the action that the targeted device is expected to undertake upon receipt of the packet.

Ping

The "Ping" instruction in the Dynamixel XH540-W150-T/R communication protocol serves to verify that an Instruction Packet has successfully reached its intended device by matching the Packet ID with the device's ID.

Read

The "Read" instruction in the Dynamixel XH540-W150-T/R communication protocol is used to request specific data from the device, enabling the controller to retrieve status or parameter values stored in the actuator's memory.

Write

The "Write" instruction in the Dynamixel XH540-W150-T/R communication protocol is employed to send specific data to the device, allowing the controller to modify parameters or settings in the actuator's memory.

Reg Write

The "Reg Write" instruction in the Dynamixel XH540-W150-T/R communication protocol is designed to place the Instruction Packet in standby status, enabling the controller to queue a specific command that is later executed when an "Action" command is received.

Action

The "Action" instruction in the Dynamixel XH540-W150-T/R communication protocol triggers the execution of a previously queued command that was placed in standby status using the "Reg Write" instruction.

Factory Reset

The "Factory Reset" instruction in the Dynamixel XH540-W150-T/R communication protocol is utilized to restore the Control Table to its original factory default settings, effectively reverting any customized configurations.

Reboot

The "Reboot" instruction in the Dynamixel XH540-W150-T/R communication protocol serves to restart the device, effectively reinitializing its internal systems without altering the Control Table settings.

Clear
The "Reset" instruction in the Dynamixel XH540-W150-T/R communication protocol is used to selectively reset specific parameters or information in the device's Control Table, allowing for targeted reconfiguration without a full factory reset.

Control Table Backup
The "Control Table Backup" instruction in the Dynamixel XH540-W150-T/R communication protocol allows for the current Control Table status data to be saved to a designated backup area or for the EEPROM data to be restored, facilitating data preservation and recovery.

Status (Return)
The "Status Packet" in the Dynamixel XH540-W150-T/R communication protocol serves as the return packet following an Instruction Packet. It provides feedback from the device, indicating the success or failure of the executed instruction and offering additional diagnostic information such as error codes or requested data.

Sync Read
The "Sync Read" instruction in the Dynamixel XH540-W150-T/R communication protocol allows for simultaneous data retrieval from multiple devices. This command reads data of the same length from the same address across several devices in a single instruction.

Sync Write
The "Sync Write" instruction in the Dynamixel XH540-W150-T/R communication protocol enables the simultaneous writing of data to multiple devices. This command allows for the uniform modification of parameters or settings at the same address and of the same length across multiple actuators in a single operation.

Fast Sync Read
The "Fast Sync Read" instruction in the Dynamixel XH540-W150-T/R communication protocol is specifically designed for high-speed, simultaneous data retrieval from multiple devices.

Bulk Read
The "Bulk Read" instruction in the Dynamixel XH540-W150-T/R communication protocol enables the simultaneous reading of data from multiple devices, each having different addresses and data lengths.

Bulk Write
The "Bulk Write" instruction in the Dynamixel XH540-W150-T/R communication protocol allows for the simultaneous writing of data to multiple devices, each with potentially different target addresses and data lengths.

Fast Bulk Read
The "Fast Bulk Read" instruction in the Dynamixel XH540-W150-T/R communication protocol is designed for expedited, simultaneous data retrieval from multiple devices, even when these devices have different target addresses and data lengths.

Parameter

In the Dynamixel XH540-W150-T/R communication protocol, the Parameters field serves as auxiliary data within the Instruction Packet, and its function varies depending on the specific instruction being executed.

CRC

In the Dynamixel XH540-W150-T/R communication protocol, the 16-bit Cyclic Redundancy Check (CRC) field serves as an integrity verification mechanism to detect any packet corruption during transmission.

Status packet

In the Dynamixel XH540-W150-T/R communication protocol, the Status Packet serves as the device's response mechanism, sent back to the main controller after receiving an Instruction Packet. The architecture of the Status Packet closely mirrors that of the Instruction Packet, with the notable addition of an ERROR field. This added field allows for error reporting.

Software

Understanding the underlying code is essential for both the optimization of current functionalities and the implementation of additional features. The code is written in C++ and is part of the AP_RobotisServo class, focusing on the *update()* function, which acts as a central controller for servo management. This method is responsible for initial setup, real-time servo monitoring, and dynamic reconfiguration, among other key responsibilities.

```
void AP_RobotisServo::update()
```

Figure 1: Function Definition for the update() method

The *update()* function serves as the central driver within the AP_RobotisServo class, effectively functioning as its main method. Designed as a void function with no parameters, it neither accepts input arguments nor returns any values.

```
if (!initialised) {
    initialised = true;
    init();
    last_send_us = AP_HAL::micros();
    DEBUG_PRINT("Not initialized, returning\n");
    return;
}
```

Figure 2: Initialization check

This code snippet evaluates the initialization status of the system. If it finds the system uninitialized, it sets the *initialized* flag to true and calls the *init()* function. The *init()* function serves as the constructor for the file, laying the groundwork for the entire program's operation. Following this, the code records the current time in microseconds in the *last_send_us variable* and exits the function early, outputting a debug print message.

```
if (port == nullptr) {
    DEBUG_PRINT("Port is nullptr, returning\n");
    return;
}
```

Figure 3: Check for Null Pointer

This code block checks the state of the *port* variable to determine if it is set to a null pointer. If the condition is met, the function terminates prematurely and logs an error message for evaluation.

```
read_bytes();

if (servo_mask != 0) {
    for (uint8_t i = 0; i < NUM_SERVO_CHANNELS; i++) {
        if (((1U << i) & servo_mask) == 0) {
            continue;
        }
        read_bytes();
    }
}
```

Figure 4: Read Bytes

Next, the *read_bytes* method is invoked, tasked with reading incoming data bytes from a serial port into a designated buffer. This ensures that the incoming data packets adhere to a specific communication protocol. Following this, the code examines the *servo_mask* to check if any servos are active. If so, it iterates through each servo channel. When a channel aligns with the *servo_mask*, the *read_bytes()* method is called specifically for that channel.

```
if (delay_cycles > 0){
    char buffer[256];
    sprintf(buffer, "\tdelay_cycles: %d, returning and waiting\n", delay_cycles);
    DEBUG_PRINT(buffer);
    // waiting for last send to complete
    delay_cycles--;
    return;
} else {
    char buffer[256];
    sprintf(buffer, "\tdelay_cycles: %d\n", delay_cycles);
    DEBUG_PRINT(buffer);
}
```

Figure 5: Delay Cycle

This section of the code manages the delay mechanism based on the cycle count and it's for asynchronous and timed behaviors.

```
uint32_t now = AP_HAL::micros();
if (last_send_us != 0 && now - last_send_us < delay_time_us) {
    // waiting for last send to complete
    return;
}
```

Figure 6: Update Timings Check

This part checks whether enough time has passed since the last update (*last_send_us*). If not, enough time has passed, the function returns early. This is to prevent the servos from overloading with too many commands in a short period.

```
if (detection_count < DETECT_SERVO_COUNT) {
    detection_count++;
    detect_servos();
    return;
}
```

Figure 7: Servo Detection

This segment ensures the controller accurately recognizes all available servos. If the detected count falls below a predefined threshold, the detection count is incremented, followed by calling the *detect_servos()* function to continue the identification process.

```
if (servo_mask == 0) {
    return;
}
```

Figure 8: Servo Mask

This code segment checks the value of *servo_mask* to determine if it is zero. If it is, the function terminates early. The *servo_mask* acts as a bitmask to identify which servos are active and should be operated upon; a zero value indicates no active servos, rendering further processing unnecessary.

```
if (configured_servos < CONFIGURE_SERVO_COUNT) {
    configured_servos++;
    last_send_us = now;
    configure_servos();
    return;
}
```

Figure 9: Configuration of Servos

This segment configures any unconfigured servos if their count is below a predetermined limit, ensuring all are set up properly before initiating further operations.

```
if (!throttle_configured && manager->throttle_servo_enable){
    configure_throttle_servo();
    throttle_configured = true;
    return;
}

if (!Watchdog_bus_configured && manager->watchdog_bus_enable){
    configure_watchdog_bus();
    Watchdog_bus_configured = true;
    return;
}
```

Figure 10: Configure Throttle & Watchdog Timer

This code section configures features such as throttle and watchdog if they are enabled but not yet set up, providing the user with the flexibility to dynamically manage these functionalities.

```
if (manager->time_to_write(port_index)) {
    write_servos();
    char buffer[256];
    sprintf(buffer, "\tThe Write timer is %d ms. The delay timer is: %d ms\n",(AP_HAL::millis() - write_timer), (delay_time_us/1000) );
    // Romaeris code to determine sample time
    //printf("\tThe writing sample time is %d ms. The delay timer is: %d ms ###### \n",(AP_HAL::millis() - write_timer), (delay_time_us/1000) );
    DEBUG_DEVELOPER_PRINT((buffer));
    write_timer =AP_HAL::millis();
}

if (manager->time_to_read(port_index)) {
    //printf("Reading Servos \n");
    read_servos();
    char buffer[256];
    sprintf(buffer, "\tThe Read timer is %d ms. The delay timer is: %d ms\n",(AP_HAL::millis() - read_timer), (delay_time_us/1000) );
    // Romaeris code to determine sample time
    //std::cout << "The Read timer is (ms)";
    //std::cout << AP_HAL::millis() - read_timer;
    //printf("\tThe Read timer is %d ms. The delay timer is: %d ms\n",(AP_HAL::millis() - read_timer), (delay_time_us/1000) );
    DEBUG_DEVELOPER_PRINT((buffer));
    read_timer =AP_HAL::millis();
}
```

Figure 11: Time to Read and Write

The code segment invokes the *write_servos()* and *read_servos()* functions based on specific timing conditions, optimizing the read-write cycles.

Problem

The primary issue at hand is an overload error, which is triggered when a sustained load exceeds the device's maximum output capabilities. Upon detection of this error, the system clears and zeroes out the torque enable and motor output, effectively locking the servo in a state where only a limited set of commands can be executed.

Solution

To address the overload issue, there are two possible solutions. The first option is to power cycle the servo, which is not feasible when the device is airborne. The alternative is to send a REBOOT command to reset the system and alleviate the overload condition.

To send the REBOOT command through the code, follow these steps:

1. Inspect incoming status packets for the presence of an error message.
    a. If no error exists, continue regular processing.
2. If an error is identified, issue a READ command to the Hardware Error Register (Address 70).
3. Determine the type of error encountered.
4. If the error is classified as an overload, execute a READ command to the Operating Mode Register (64) and store the value.
5. Conduct a READ command to capture the last safe value before the overload for the specific Operating Mode.
6. Transmit a REBOOT command to the servo.
7. Issue a WRITE command to restore the previously stored Operating Mode to the servo (Address 11).
8. Execute a WRITE command to re-enable torque by setting a '1' in the Torque Enable Register.
9. Restore the safe operating value to the appropriate register corresponding to the Operating Mode.

The following state diagram outlines the steps required to correct the overload error.

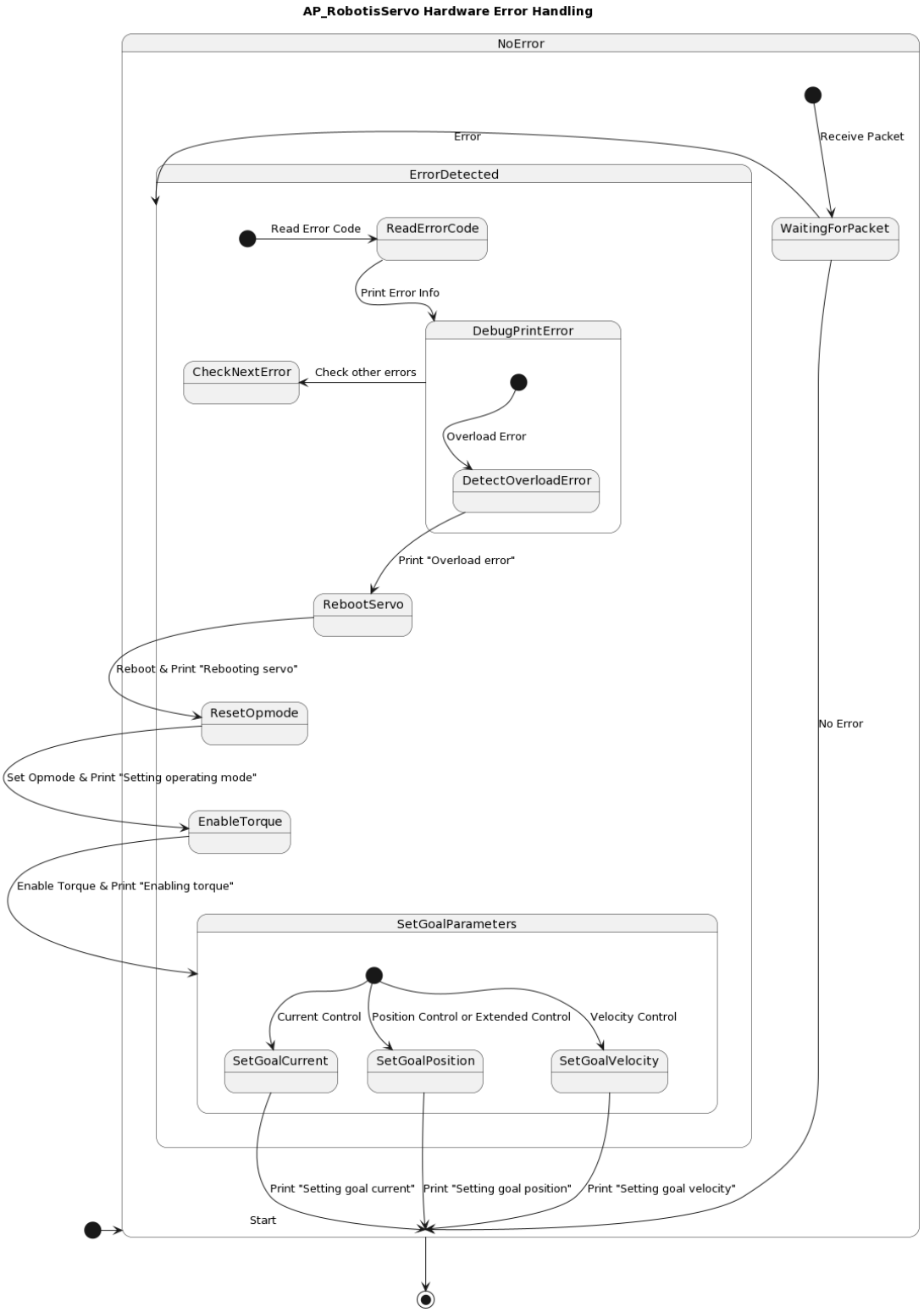


Figure 12: State Diagram for resolving the overload error.

Testing

Modified Functions

```
/*
    send a specified instruction to a single servo
    @param id: servo id
    @param inst: instruction
    @param reg: register address
    @param value: value to write
    @param len: length of value

    @return: void
 */
void AP_RobotisServo::send_command(uint8_t id, uint8_t inst ,uint16_t reg, uint32_t value, uint8_t len)
{

    //since this function was rewritten to be modular, we need to check
    //to add any read commands to the last_read_send array
    if(inst == INST_READ)
    {
        DEBUG_DEV_PRINT("THE last read is  %d \n", reg);
        last_read_send[(unsigned int)id -1] = reg;
    }

    uint8_t txpacket[16] {};
    txpacket[PKT_ID] = id;
    txpacket[PKT_LENGTH_L] = 5 + len;
    txpacket[PKT_LENGTH_H] = 0;
    txpacket[PKT_INSTRUCTION] = inst;
    txpacket[PKT_INSTRUCTION+1] = DXL_LOBYTE(reg);
    txpacket[PKT_INSTRUCTION+2] = DXL_HIBYTE(reg);
    memcpy(&txpacket[PKT_INSTRUCTION+3], &value, MIN(len,4));


    send_packet(txpacket);
}
```

Figure 13: Modified Send Command

Originally, distinct functions were designated for each send command instruction. I have since restructured the *send_command()* function to accept the instruction type as a parameter, allowing it to be more modular. This modification has proven effective; during testing, READs and WRITEs executed flawlessly within the expected timeframe, resulting in smooth servo movement.

```
AP_RobotisServo::AP_RobotisServo(AP_RobotisManager *manager_ptr, uint8_t port_index_arg) :
port_index(port_index_arg)
{
    // set defaults from the parameter table
    //AP_Param::setup_object_defaults(this, var_info);

    manager = manager_ptr;
    last_send_us = 0;
    delay_time_us = 0;
    servo_mask = 0;
    detection_count = 0;
    configured_servos = 0;
    initialised = false;
    memset(pktbuf, 0, sizeof(pktbuf));
    pktbuf_ofs = 0;
    connected_servos = 0;
    throttle_configured = 0;
    Watchdog_bus_configured = 0;
    read_timer = 0;
    write_timer = 0;
    delay_cycles = 0;
    last_current_time = 0;
    last_position_time = 0;
    last_velocity_time = 0;
    current_read_delay = 0;
    position_read_delay = 0;
    velocity_read_delay = 0;

    checked_opmode = 0;
    std::fill_n(opmode_set, NUM_SERVO_CHANNELS, 3);
    std::fill_n(torque_set, NUM_SERVO_CHANNELS, 0);
    std::fill_n(last_read_send, NUM_SERVO_CHANNELS, 0);
    std::fill_n(present_positions, NUM_SERVO_CHANNELS, 0);
    std::fill_n(present_current, NUM_SERVO_CHANNELS, 0);
    std::fill_n(present_velocity, NUM_SERVO_CHANNELS, 0);
    servos_all_configured = false;

    //Init for the reading parameters check
    check_cur = true;
    check_pos = true;
    check_vel = true;

    check_error = false;


}
```

Figure 14: Constructor for AP_RobotisServo

I introduced a *check_error* flag in the constructor to facilitate future error monitoring and management.

In the *process_packet()* function (refer to the appendix's *process_packet()* section), I've introduced a segment of code that forwards the current packet being processed to the *detect_hardware_error()* function for error evaluation. Additionally, a debugging section has been implemented to verify if a READ instruction was specifically directed to read the hardware error register. A separate section has also been added to capture the current error code when the READ instruction accesses the hardware error register and the check_error flag is set to true.

In the *read_servos()* (refer to section *read_servos()* in the appendix) I have incorporated a condition to evaluate the *check_error* flag. If the flag is set to true, the *handle_hardware_errors()* function is invoked for subsequent error handling. I also modified the subsequent if statements in the function so that if the *check_error* flag is true no other READs are sent.

New Additions

```
#define REG_HARDWARE_ERROR_STATUS 70
#define  ERRBIT_VOLTAGE      0x01 // Bit 0
// Bit 1 is unused and always '0', so no define is needed for it.
#define  ERRBIT_OVERHEAT       0x04 // Bit 2
#define  ERRBIT_MOTOR_ENCODER     0x08 // Bit 3
#define  ERRBIT_ELECTRIC_SHOCK    0x10 // Bit 4
#define  ERRBIT_OVERLOAD          0x80 // Bit 5
// Bit 6 and Bit 7 are unused and always '0', so no defines are needed for them.
```

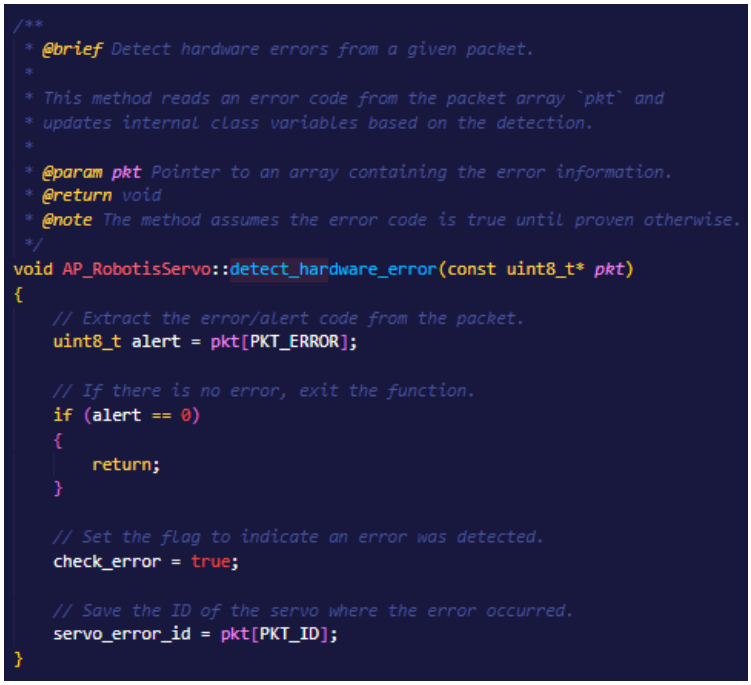
Figure 15: Hardware Error Defines

To enhance code readability and for future modifications, define statements have been introduced for both the hardware register and the different error bits. These defined statements are constants, making it easier to identify and manage specific register addresses and error conditions within the code.

```
//another Developer's debug macros but this allows
//different formats & various arguments
#define DEV_DEBUG true
#if DEV_DEBUG == true
#define DEBUG_DEV_PRINT(fmt, ...) printf(fmt, ##__VA_ARGS__)
#else
#define DEBUG_DEV_PRINT(fmt, ...) do {} while (0)
#endif
```

Figure 16: New Developer Debug Macro

A new developer debug macro has been introduced to enhance diagnostic capabilities. This updated feature supports various format specifiers including hexadecimal and character types, offering flexibility beyond the previous version which was limited to string-only inputs. To utilize this feature, the desired format must be explicitly specified within the macro. This allows developers to directly print complete variables, avoiding the need for the built-in sprintf() function. The original developer debug print functionality is preserved in the code.



```
/**
 * @brief Detect hardware errors from a given packet.
 *
 * This method reads an error code from the packet array `pkt` and
 * updates internal class variables based on the detection.
 *
 * @param pkt Pointer to an array containing the error information.
 * @return void
 * @note The method assumes the error code is true until proven otherwise.
 */
void AP_RobotisServo::detect_hardware_error(const uint8_t* pkt)
{
    // Extract the error/alert code from the packet.
    uint8_t alert = pkt[PKT_ERROR];

    // If there is no error, exit the function.
    if (alert == 0)
    {
        return;
    }

    // Set the flag to indicate an error was detected.
    check_error = true;

    // Save the ID of the servo where the error occurred.
    servo_error_id = pkt[PKT_ID];
}
```

Figure 17: detect_hardware_error() function

This function is designed to identify errors within a given packet. Upon receiving a pointer to a packet, the code reads its ERROR field. If the packet is error-free, the code proceeds as usual. However, if an error is detected, the *check_error* flag is activated, and the id of the servo exhibiting the error is stored in the *servo_error_id* variable.

The final addition to the code is the *handle_hardware_errors()* function (refer to the *handle_hardware_errors()* function in the appendix), which is tasked with managing hardware errors. Initially, a READ command is dispatched to the hardware error register to retrieve the error status and identify the servo that triggered the error. Subsequently, the function evaluates the nature of the error; in the event of an overload error, the current operating mode is retrieved. Following this, a REBOOT command is issued. Once the system reinitializes, the original operating mode is restored, torque is re-enabled, and the last safe value is set as the current value for the designated operating mode.

Conclusion

In conclusion, this report has comprehensively analyzed the modifications and enhancements made to the *AP_RobotisServo.cpp*. Primarily focusing on error detection and handling capabilities. The introduction of a modular *send_command()* function, an advanced developer debug macro, has significantly elevated the code's robustness and diagnostic proficiency. Moreover, the addition of the *detect_hardware_error()* and *handle_hardware_errors()* functions has initiated a robust framework for error management, setting the stage for further improvements and system optimization.

Appendix

process_packet()

```
void AP_RobotisServo::process_packet(const uint8_t* pkt, uint8_t length)
{
    uint8_t id = pkt[PKT_ID];
    if (id > 16 || id < 1) {
        // discard packets from servos beyond max or min. Note that we
        // don't allow servo 0, to make mapping to SERVOn_* parameters
        // easier
        return;
    }
    uint16_t id_mask = (1U << (id - 1));
    if (!(id_mask & servo_mask)) {
        // mark the servo as present
        servo_mask |= id_mask;
        connected_servos++;
        printf("Robotis: new servo %u\n", id);
    }
    uint8_t message_size = pkt[PKT_LENGTH_L];
    //printf("Robotis: incoming message with ID: %u. The is before checking the message size. Current size is: %u\n", id, message_size);

    char print_buffer[256];
    sprintf(print_buffer, "\tThe packet being received from the port: \n\t");
    for (int j = 0; j < length; j++) {
        sprintf(print_buffer + strlen(print_buffer), " %x ", pkt[j]);
    }
    sprintf(print_buffer + strlen(print_buffer), " \n ");

    DEBUG_DEV_PRINT("%s", print_buffer );
    //DEBUG_DEV_PRINT("The length of the print_buffer is: %lu\n", (unsigned long)strlen(print_buffer));

    //int present_position = 0;
    //int current = 0;
    uint8_t one_byte_data = 0;
    uint16_t incoming_status = last_read_send[(unsigned int)id - 1];
    //uint32_t three_byte_data = 0;

    DEBUG_DEV_PRINT("Incoming status: %x\n\n", incoming_status);
```

```
detect_hardware_error(pkt);
switch (incoming_status) {
case REG_PRESENT_POSITION:
    DEBUG_PRINT(("Last read command send was for position\n"));
    break;

case REG_PRESENT_CURRENT:
    DEBUG_PRINT(("Last read command send was for current\n"));
    break;

case REG_PRESENT_VELOCITY:
    DEBUG_PRINT(("Last read command send was for velocity\n"));
    break;

case REG_OPERATING_MODE:
    DEBUG_PRINT(("Last read command send was for Operating mode\n"));
    checked_opmode = 1;
    break;

case REG_TORQUE_ENABLE:
    DEBUG_PRINT(("Last read command send was for read the status of the torque\n"));
    break;

case REG_HARDWARE_ERROR_STATUS:
    DEBUG_DEV_PRINT("Last read command send was for read the status of the hardware error\n");
    break;

default:
    //char buffer[256];
    sprintf(print_buffer, "Reading from ID: %d at Register address of %u \n", (unsigned int)id, incoming_status);
    DEBUG_PRINT((print_buffer));
    break;
}
```

```
case 5:
    one_byte_data = pkt[PKT_INSTRUCTION + 2];
    //char buffer[256];
    sprintf(print_buffer, "Robotis: incoming message with ID: %u. one byte of data: %u. \n", id, one_byte_data);
    DEBUG_PRINT((print_buffer));
    if (incoming_status == REG_OPERATING_MODE) {
        //Check if right opmode has been set or not
        // Get a pointer to the channel
        SRV_Channel* c = SRV_Channels::srv_channel(id - 1);
        if (c != nullptr) {
            const uint8_t opmode = c->get_opmode();
            sprintf(print_buffer, "Robotis: incoming message with ID: %u. one byte of data: %u. Should be: %u \n", id, one_byte_data, opmode);
            DEBUG_PRINT((print_buffer));

            if (opmode == one_byte_data) {

                //checked_opmode = 1;
                //Error on the next line
                opmode_set[id] = opmode;
                //printf("The opmode has been checked and confirmed for this ID: %u Opmode_set: %u\n", id, opmode_set[id]);
            }
        }
    }

    if ((incoming_status == REG_TORQUE_ENABLE) && (checked_opmode == 1) && (one_byte_data == 1)) {
        //This servo is set and ready for commands
        torque_set[id] = 1;

    }

    if (check_error && incoming_status == REG_HARDWARE_ERROR_STATUS)
    {
        Error_code = -1;
        //DEBUG_DEV_PRINT("==========================================");
    }

    break;

    //6 = response from sending read request to the present current
case 6:

    process_packet_current(pkt, id);
    break;

//7 = response from a ping request
case 7:
    //three_byte_data = (pkt[PKT_INSTRUCTION + 3] << 8) | pkt[PKT_INSTRUCTION + 2];
    sprintf(print_buffer, "Robotis: incoming message: Received a ping from Dynamixel ID: %u.  \n", id);
    DEBUG_PRINT((print_buffer));
    break;
```

```
//8 = response from sending read request to the present position
case 8:
    if(incoming_status == REG_PRESENT_POSITION)
    {
        process_packet_position(pkt, id);
    }
    else if(incoming_status == REG_PRESENT_VELOCITY)
    {
        process_packet_velocity(pkt, id);
    }
    else if(incoming_status == REG_HARDWARE_ERROR_STATUS)
    {
        DEBUG_DEV_PRINT("RESPONSE Packet: ");

        for(unsigned int i = 0; i < sizeof(pkt); ++i)
        {
            DEBUG_DEV_PRINT("%02X ", pkt[i]);
        }
        DEBUG_DEV_PRINT("\n");

        Error_code = pkt[PKT_PARAMETER0];
        break;
    }

    break;

}
if (manager->logging_enabled && (message_size == 6 || message_size == 8)){
    char *topic = log_topics[id-1]; //subract 1 because topics start at 1 instead of 0
    sprintf(print_buffer, "Logging to %s\n", topic);
    DEBUG_PRINT(print_buffer);
    SRV_Channel* c = SRV_Channels::srv_channel(id - 1);
    AP::logger().Write(topic, "TimeUS,position,pos_delay,current,cur_delay,velocity,vel_delay","Qiiiiii",
            AP_HAL::micros64(),
            c->get_present_position(),
            position_read_delay,
            c->get_present_current(),
            current_read_delay,
            c->get_present_velocity(),
            velocity_read_delay);
}
```

read_servos()

```
if(check_error)
{
    check_error = false;
    handle_hardware_errors();
}
```

```
if((read_parameter & CURRENT_BITMASK)&& !check_error) {
    //The check_cur is to make sure only one current syncRead packet is sent out for this time read_servos() is called.
    //Once the current syncRead packet is sent it gives a chance for a synceRead for Position or Velocity to be sent next time read_servos() is called.
    //round_check to to make sure no other syncRead packet is created besides this one for current
    if((check_cur) && (!round_check)){
        DEBUG_PRINT(("============================================================Reading the Current=============================================================\n"));
        auto now = AP_HAL::millis();
        if (last_current_time == 0){
            last_current_time = now;
        } else {
            current_read_delay = now - last_current_time;
            last_current_time = now;
            sprintf(timer_buffer, "\t current reading delay: %d ms\n", current_read_delay);
            DEBUG_PRINT(timer_buffer);
        }
        device_reg_arr[device_num] = read_reg_by_mode[OPMODE_CURR_CONTROL];
        device_data_length_arr[device_num] = data_len_by_mode[OPMODE_CURR_CONTROL];
        round_check = true;
        check_cur = false;
    }

    //increment read counter for delay
    read_counter++;

} else {
    check_cur = false;
}
```

```
if((read_parameter & POSITION_BITMASK)&& !check_error){
    //The check_pos is to make sure only one position syncRead packet is sent out for this time read_servos() is called.
    //Once the position syncRead packet is sent it gives a chance for a synceRead for Velocity to be sent next time read_servos() is called.
    //round_check to to make sure no other syncRead packet is created besides this one for position
    if((check_pos) && (!round_check)){
        DEBUG_PRINT(("============================================================Reading the position=============================================================\n"));
        auto now = AP_HAL::millis();
        if (last_position_time == 0){
            last_position_time = now;
        } else {
            position_read_delay = now - last_position_time;
            last_position_time = now;
            sprintf(timer_buffer, "\t position reading delay: %d ms\n", position_read_delay);
            DEBUG_PRINT(timer_buffer);
        }
        device_reg_arr[device_num] = read_reg_by_mode[OPMODE_POS_CONTROL];
        device_data_length_arr[device_num] = data_len_by_mode[OPMODE_POS_CONTROL];
        round_check = true;
        check_pos = false;
    }

    //increment read counter for delay
    read_counter++;
} else {
    check_pos = false;
}
```

```
if((read_parameter & VELOCITY_BITMASK)&& !check_error){
    //The check_vel is to make sure only one velocity syncRead packet is sent out for this time read_servos() is called.
    //Once the velocity syncRead packet is sent, then next time read_servos() is called.
    //round_check to to make sure no other syncRead packet is created besides this one for position
    if((check_vel) && (!round_check)){
        DEBUG_PRINT(("==========================================================Reading the Velocity==========================================================\n"));
        auto now = AP_HAL::millis();
        if (last_velocity_time == 0){
            last_velocity_time = now;
        } else {
            velocity_read_delay = now - last_velocity_time;
            last_velocity_time = now;
            sprintf(timer_buffer, "\t velocity reading delay: %d ms\n", velocity_read_delay);
            DEBUG_PRINT(timer_buffer);
        }
        DEBUG_PRINT(buffer);
        device_reg_arr[device_num] = read_reg_by_mode[OPMODE_VEL_CONTROL];
        device_data_length_arr[device_num] = data_len_by_mode[OPMODE_VEL_CONTROL];
        round_check = true;
        check_vel = false;
    }

    //increment read counter for delay
    read_counter++;
} else {
    check_vel = false;

}
```

```
if((!check_cur && !check_pos) && !check_error)
    check_cur = true;
    check_pos = true;
    check_vel = true;
}
```

handle_hardware_errors()

```
void AP_RobotisServo:: handle_hardware_errors(void)
{

    uint8_t id = servo_error_id;
    uint8_t error_code = Error_code;
    send_command(id, INST_READ, REG_HARDWARE_ERROR_STATUS, 1, 2);
    //check_error = false;


    DEBUG_DEV_PRINT("Hardware error detected at device %d\n", id);
    DEBUG_DEV_PRINT("Error code: %02x\n", error_code);

    //for Later implementation
    if(error_code & ERRBIT_VOLTAGE)
    {
        DEBUG_DEV_PRINT("Input voltage error\n");
    }

    if(error_code & ERRBIT_OVERHEAT)
    {
        DEBUG_DEV_PRINT("Overheat error\n");
    }

    if(error_code & ERRBIT_MOTOR_ENCODER)
    {
        DEBUG_DEV_PRINT("Motor encoder error\n");
    }

    if(error_code & ERRBIT_ELECTRIC_SHOCK)
    {
        DEBUG_DEV_PRINT("Electric shock error\n");
    }
```

```
//overload
if(error_code & ERRBIT_OVERLOAD)
{

    DEBUG_DEV_PRINT("Overload error\n");
    DEBUG_DEV_PRINT("ID: %d\n", id);//id of the overloaded servo

    //get the opmode of the overloaded servo
    SRV_Channel* c = SRV_Channels::srv_channel(id);
    uint8_t opmode = c->get_opmode();
    DEBUG_DEV_PRINT("Current operating mode: %d\n", opmode);

    //send the reboot command to the overloaded servo
    send_command(id, INST_REBOOT, 1);
    DEBUG_DEV_PRINT("Rebooting servo\n");

    //send the reboot command to the overloaded servo
    send_command(id, INST_WRITE, REG_OPERATING_MODE, opmode, 1);

    //when the reboot command is sent it will
    //reset the opmode as well the torque enable,
    //goal position, goal velocity, goal current


    //reset the opmode
    DEBUG_DEV_PRINT("Setting operating mode to %02x\n", opmode);

    send_command(id, INST_WRITE, REG_TORQUE_ENABLE, 1, 1);
    DEBUG_DEV_PRINT("Enabling torque\n");

    //This section is to set the goal position, goal velocity, goal current to the last
    //safe value that was sent to the servo before the overload error occured

    // if (opmode == OPMODE_CURR_CONTROL)
    // {
    //     send_command(id, INST_WRITE, REG_GOAL_CURRENT, get_present_current(id), 2);
    //     DEBUG_DEV_PRINT("Setting goal current to \n");
    // }
    // else if (opmode == OPMODE_POS_CONTROL || opmode == OPMODE_EXT_POS_CONTROL)
    // {
    //     send_command(id, INST_WRITE, REG_GOAL_POSITION, get_present_postion(id), 4);
    //     DEBUG_DEV_PRINT("Setting goal position to \n");
    // }
    // else if(opmode == OPMODE_VEL_CONTROL)
    // {
    //     send_command(id, INST_WRITE, REG_GOAL_VELOCITY, get_present_velocity(id), 2);
    //     DEBUG_DEV_PRINT("Setting goal position to \n");
    // }


}
```