DRWIG Parser Integration Documentation

This document provides comprehensive documentation for the DRWIG protocol parser integration, including serial communication, TLV format parsing, and data conversion mechanisms.

📋 Table of Contents

Overview

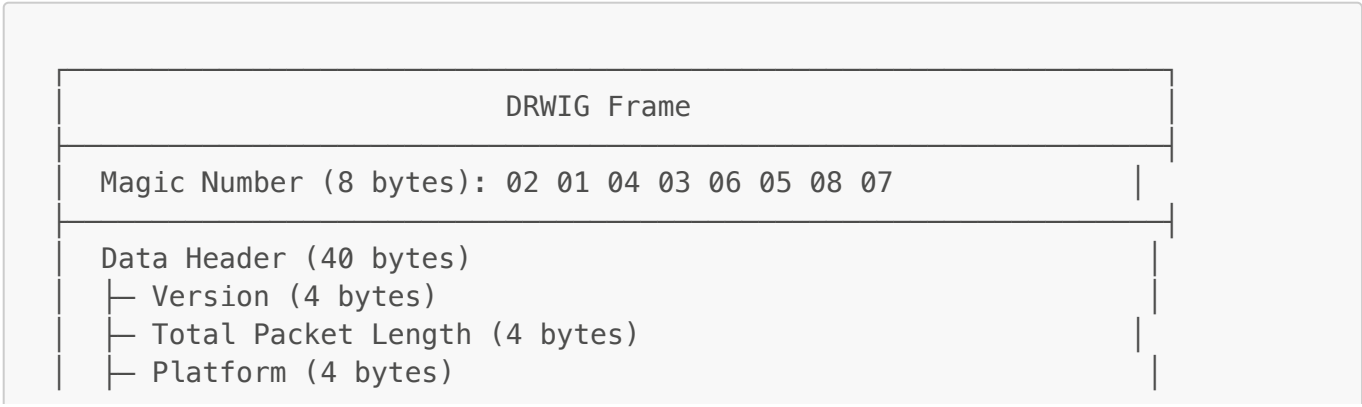
The DRWIG Parser Integration module provides real-time communication with DRWIG radar hardware via UART/Serial interfaces. It implements the DRWIG TLV (Type-Length-Value) protocol for parsing radar data frames containing detected objects, clusters, and tracked objects.

Key Features

- **Real-time UART Communication**: High-speed serial data acquisition
- **TLV Protocol Parsing**: Robust parsing of Type-Length-Value formatted data
- **Multi-threaded Architecture**: Separate threads for reading and parsing
- **Thread-safe Data Structures**: Lock-free queues for high-performance data flow
- **Callback-based Interface**: Event-driven data delivery to the visualizer
- **Error Recovery**: Robust error handling and connection recovery

DRWIG Protocol Structure

Frame Format



```
│   ├─ Frame Number (4 bytes)                                          │
│   ├─ Time CPU Cycles (4 bytes)                                       │
│   ├─ Number of Detected Objects (4 bytes)                            │
│   ├─ Number of TLVs (4 bytes)                                        │
│   └─ Sub Frame Number (4 bytes)                                      │
├──────────────────────────────────────────────────────────────────────┤
│  TLV Block 1                                                         │
│   ├─ TLV Header (8 bytes)                                            │
│   │   ├─ Type (4 bytes)                                              │
│   │   └─ Length (4 bytes)                                            │
│   ├─ Object Header (4 bytes)                                         │
│   │   ├─ Number of Objects (2 bytes)                                 │
│   │   └─ Format (2 bytes)                                            │
│   └─ Object Data (Variable length)                                   │
├──────────────────────────────────────────────────────────────────────┤
│  TLV Block 2 (Optional)                                              │
│   └─ ... (Same structure as TLV Block 1)                             │
├──────────────────────────────────────────────────────────────────────┤
│  TLV Block N (Optional)                                              │
│   └─ ... (Additional TLV blocks)                                     │
└──────────────────────────────────────────────────────────────────────┘
```

Magic Number Sequence

The DRWIG protocol uses a specific 8-byte magic number sequence to identify frame boundaries:

```
constexpr std::array<uint8_t, kDrwigDataMagicNumberSize> kMagicNumber {
    2, 1, 4, 3, 6, 5, 8, 7
};
```

This sequence provides reliable frame synchronization and helps recover from data corruption or loss.

DrwigParser Class

Class Declaration

File: `include/drwig_parser.hpp`

```
class DrwigParser {
public:
    using Frame_t = std::vector<uint8_t>;

    DrwigParser(std::string& serial_port, int baud_rate);
    ~DrwigParser();

    // Main interface
    void Init();                    // Initialize parser and start threads
    void Parse();                   // Start parsing (blocking)
    void Wait();                    // Wait for completion
```

```
    // Callback registration
    void RegisterObjectHandler(DrwigDetectObjectHandler callback);
    void RegisterObjectHandler(DrwigClusterObjectHandler callback);
    void RegisterObjectHandler(DrwigTrackedObjectHandler callback);

private:
    // Internal methods
    void ParseData();
    void ParseDataImpl();
    void ReadUartImpl();
    void OpenComPort();

    // Protocol parsing
    bool isMagicNumber();
    void readMagicNumber();
    bool readHeader(Frame_t& frame);
    bool readTlvHeader(Frame_t& frame);
    bool readObjectHeader(Frame_t& frame, DrwigObjectHeader& obj);
    bool readDetectedObject(Frame_t& frame);
    bool readClusterObject(Frame_t& frame);
    bool readTrackedObject(Frame_t& frame);

    // State and data
    DrwigDataHeader m_drwig_data_header;
    DrwigTlvHeader m_drwg_tlv_header;
    DrwigDataMagicNumber m_magic_number;

    // Object storage
    std::vector<DrwigDetectedObject> m_drwig_detected_object;
    std::vector<DrwigClusterObject> m_drwig_cluster_object;
    std::vector<DrwigTrackedObject> m_drwig_tracked_object;

    // Threading and synchronization
    std::thread read_uart;
    std::thread parse_data;
    TSQueue<uint32_t> write_queue;
    TSQueue<uint32_t> reader_queue;
    std::mutex enter_m_mutex;
    std::mutex buffer_m_mutex;

    // Serial communication
    SerialPort m_serial_object;
    std::string serial_port;
    int baud_rate;
    size_t ms_timeout = 250;

    // Callbacks
    DrwigDetectObjectHandler drwig_detected_object_handler;
    DrwigClusterObjectHandler drwig_cluster_object_handler;
    DrwigTrackedObjectHandler drwig_tracked_object_handler;
};
```

Initialization Process

```
DrwigParser::DrwigParser(std::string& a_serial_port, int a_baud_rate)
    : serial_port(a_serial_port), baud_rate(a_baud_rate), data_idx(0),
read_thread(true) {
    // Initialize buffers
    m_buffer.resize(2);
    for (auto& buffer : m_buffer) {
        buffer.reserve(max_frame_size);
    }
}

void DrwigParser::Init() {
    // Open serial port
    OpenComPort();

    // Start reader thread
    read_uart = std::thread(&DrwigParser::ReadUartImpl, this);

    // Start parser thread
    parse_data = std::thread(&DrwigParser::ParseDataImpl, this);

    std::cout << "DRWIG Parser initialized with port: " << serial_port
              << " at " << baud_rate << " baud" << std::endl;
}

void DrwigParser::OpenComPort() {
    try {
        m_serial_object.Open(serial_port);
        m_serial_object.SetBaudRate(static_cast<BaudRate>(baud_rate));
        m_serial_object.SetCharacterSize(CharacterSize::CHAR_SIZE_8);
        m_serial_object.SetFlowControl(FlowControl::FLOW_CONTROL_NONE);
        m_serial_object.SetParity(Parity::PARITY_NONE);
        m_serial_object.SetStopBits(StopBits::STOP_BITS_1);

        std::cout << "Serial port " << serial_port << " opened
successfully" << std::endl;

    } catch (const std::exception& e) {
        std::cerr << "Failed to open serial port " << serial_port
                  << ": " << e.what() << std::endl;
        throw;
    }
}
```

Data Type Definitions

Core Constants

File: `include/drwig_type.hpp`

```
constexpr uint32_t kDrwigDataMagicNumberSize{8U};
constexpr uint32_t kDrwigDataHeaderFrameSize{40U};
constexpr uint32_t kMaxDetObject{200U};
constexpr uint32_t kMaxClusterObject{24U};
constexpr uint32_t kMaxNumTracker{24U};
```

Magic Number Structure

```
struct DrwigDataMagicNumber {
    std::array<uint8_t, kDrwigDataMagicNumberSize> data;
};
```

Data Header Structure

```
struct DrwigDataHeader {
    uint32_t version;               // Protocol version
    uint32_t total_packet_len;      // Total packet length in bytes
    uint32_t platform;              // Platform identifier
    uint32_t frame_number;          // Sequential frame number
    uint32_t time_cpu_cycles;       // Timestamp in CPU cycles
    uint32_t num_detected_obj;      // Number of detected objects
    uint32_t num_tlvs;              // Number of TLV blocks
    uint32_t sub_frame_number;      // Sub-frame identifier
};
```

TLV Type Enumeration

```
enum class DrwigTlvTypes : uint32_t {
    MSG_NULL = 0,                       // Null message
    MSG_DETECTED_POINTS,                // List of detected points
    MSG_CLUSTERS,                       // Cluster information
    MSG_TRACKED_OBJ,                    // Tracked object data
    MSG_PARKING_ASSIST,                 // Parking assist data
    MSG_RANGE_DOPPLER_HEAT_MAP,         // Range/Doppler heatmap
    MSG_STATS,                          // Statistics information
    MSG_DETECTED_POINTS_SIDE_INFO,      // Side information for detections
    MSG_MAX                             // Maximum value marker
};
```

Object Data Structures

```
struct DrwigDetectedObject {
    int16_t doppler_velocity;       // Doppler velocity (fixed-point)
```

```
    uint16_t peak_value;            // Signal peak value
    int16_t x;                      // X coordinate (fixed-point)
    int16_t y;                      // Y coordinate (fixed-point)
    int16_t z;                      // Z coordinate (fixed-point)
};

struct DrwigClusterObject {
    int16_t x;                      // X coordinate (fixed-point)
    int16_t y;                      // Y coordinate (fixed-point)
    uint16_t width;                 // Width (fixed-point)
    uint16_t length;                // Length (fixed-point)
};

struct DrwigTrackedObject {
    int16_t x;                      // X coordinate (fixed-point)
    int16_t y;                      // Y coordinate (fixed-point)
    int16_t vx;                     // X velocity (fixed-point)
    int16_t vy;                     // Y velocity (fixed-point)
    uint16_t width;                 // Width (fixed-point)
    uint16_t length;                // Length (fixed-point)
};
```

TLV Protocol Implementation

TLV Header Parsing

```
bool DrwigParser::readTlvHeader(Frame_t& c_frame) {
    if (c_frame.size() < sizeof(DrwigTlvHeader)) {
        return false;
    }

    // Read TLV type and length
    std::memcpy(&m_drwg_tlv_header, c_frame.data(),
sizeof(DrwigTlvHeader));

    // Validate TLV type
    if (m_drwg_tlv_header.type >= DrwigTlvTypes::MSG_MAX) {
        std::cerr << "Invalid TLV type: " << static_cast<uint32_t>
(m_drwg_tlv_header.type) << std::endl;
        return false;
    }

    // Validate TLV length
    if (m_drwg_tlv_header.length > max_frame_size) {
        std::cerr << "TLV length too large: " << m_drwg_tlv_header.length
<< std::endl;
        return false;
    }

    return true;
}
```

Object Header Parsing

```
bool DrwigParser::readObjectHeader(Frame_t& c_frame, DrwigObjectHeader&
obj) {
    if (c_frame.size() < sizeof(DrwigObjectHeader)) {
        return false;
    }

    std::memcpy(&obj, c_frame.data(), sizeof(DrwigObjectHeader));

    // Validate object count based on type
    uint16_t max_objects = 0;
    switch (m_drwg_tlv_header.type) {
        case DrwigTlvTypes::MSG_DETECTED_POINTS:
            max_objects = kMaxDetObject;
            break;
        case DrwigTlvTypes::MSG_CLUSTERS:
            max_objects = kMaxClusterObject;
            break;
        case DrwigTlvTypes::MSG_TRACKED_OBJ:
            max_objects = kMaxNumTracker;
            break;
        default:
            max_objects = 1000; // Generic limit
            break;
    }

    if (obj.num_obj > max_objects) {
        std::cerr << "Object count exceeds limit: " << obj.num_obj
                  << " > " << max_objects << std::endl;
        return false;
    }

    return true;
}
```

Object Data Parsing

```
bool DrwigParser::readDetectedObject(Frame_t& c_frame) {
    // Read object header
    if (!readObjectHeader(c_frame, m_drwig_detected_object_hdr)) {
        return false;
    }

    // Calculate required data size
    size_t header_size = sizeof(DrwigObjectHeader);
    size_t object_size = sizeof(DrwigDetectedObject);
    size_t required_size = header_size +
```

```
(m_drwig_detected_object_hdr.num_obj * object_size);

    if (c_frame.size() < required_size) {
        std::cerr << "Insufficient data for detected objects" <<
std::endl;
        return false;
    }

    // Clear previous objects
    m_drwig_detected_object.clear();
    m_drwig_detected_object.reserve(m_drwig_detected_object_hdr.num_obj);

    // Read object data
    uint8_t* data_ptr = c_frame.data() + header_size;
    for (uint16_t i = 0; i < m_drwig_detected_object_hdr.num_obj; ++i) {
        DrwigDetectedObject obj;
        std::memcpy(&obj, data_ptr + (i * object_size), object_size);
        m_drwig_detected_object.push_back(obj);
    }

    // Call registered callback
    if (drwig_detected_object_handler) {
        drwig_detected_object_handler(m_drwig_detected_object_hdr,
m_drwig_detected_object);
    }

    return true;
}

bool DrwigParser::readClusterObject(Frame_t& c_frame) {
    if (!readObjectHeader(c_frame, m_drwig_cluster_object_hdr)) {
        return false;
    }

    size_t header_size = sizeof(DrwigObjectHeader);
    size_t object_size = sizeof(DrwigClusterObject);
    size_t required_size = header_size +
(m_drwig_cluster_object_hdr.num_obj * object_size);

    if (c_frame.size() < required_size) {
        return false;
    }

    m_drwig_cluster_object.clear();
    m_drwig_cluster_object.reserve(m_drwig_cluster_object_hdr.num_obj);

    uint8_t* data_ptr = c_frame.data() + header_size;
    for (uint16_t i = 0; i < m_drwig_cluster_object_hdr.num_obj; ++i) {
        DrwigClusterObject obj;
        std::memcpy(&obj, data_ptr + (i * object_size), object_size);
        m_drwig_cluster_object.push_back(obj);
    }

    if (drwig_cluster_object_handler) {
```

```
        drwig_cluster_object_handler(m_drwig_cluster_object_hdr,
m_drwig_cluster_object);
    }

    return true;
}

bool DrwigParser::readTrackedObject(Frame_t& c_frame) {
    if (!readObjectHeader(c_frame, m_drwig_tracked_object_hdr)) {
        return false;
    }

    size_t header_size = sizeof(DrwigObjectHeader);
    size_t object_size = sizeof(DrwigTrackedObject);
    size_t required_size = header_size +
(m_drwig_tracked_object_hdr.num_obj * object_size);

    if (c_frame.size() < required_size) {
        return false;
    }

    m_drwig_tracked_object.clear();
    m_drwig_tracked_object.reserve(m_drwig_tracked_object_hdr.num_obj);

    uint8_t* data_ptr = c_frame.data() + header_size;
    for (uint16_t i = 0; i < m_drwig_tracked_object_hdr.num_obj; ++i) {
        DrwigTrackedObject obj;
        std::memcpy(&obj, data_ptr + (i * object_size), object_size);
        m_drwig_tracked_object.push_back(obj);
    }

    if (drwig_tracked_object_handler) {
        drwig_tracked_object_handler(m_drwig_tracked_object_hdr,
m_drwig_tracked_object);
    }

    return true;
}
```

Serial Communication

UART Reading Thread

```
void DrwigParser::ReadUartImpl() {
    std::cout << "UART reader thread started" << std::endl;

    while (read_thread) {
        try {
            // Wait for data availability
            if (!m_serial_object.IsDataAvailable()) {
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
```

```
                continue;
            }

            // Read available data
            std::string received_data;
            m_serial_object.ReadLine(received_data, '\n', ms_timeout);

            if (!received_data.empty()) {
                // Process received data
                {
                    std::lock_guard<std::mutex> lock(buffer_m_mutex);

                    // Add data to current buffer
                    Frame_t& current_buffer = m_buffer[data_idx];
                    current_buffer.insert(current_buffer.end(),
                                          received_data.begin(),
                                          received_data.end());

                    // Check for complete frame
                    if (isCompleteFrame(current_buffer)) {
                        // Notify parser thread
                        reader_queue.push(data_idx);

                        // Switch to next buffer
                        data_idx = (data_idx + 1) % m_buffer.size();
                        m_buffer[data_idx].clear();
                    }
                }
            }

        } catch (const std::exception& e) {
            std::cerr << "UART read error: " << e.what() << std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    }

    std::cout << "UART reader thread stopped" << std::endl;
}
```

Frame Validation

```
bool DrwigParser::isMagicNumber() {
    if (magnumBytes[0] == kMagicNumber[0] &&
        magnumBytes[1] == kMagicNumber[1] &&
        magnumBytes[2] == kMagicNumber[2] &&
        magnumBytes[3] == kMagicNumber[3] &&
        magnumBytes[4] == kMagicNumber[4] &&
        magnumBytes[5] == kMagicNumber[5] &&
        magnumBytes[6] == kMagicNumber[6] &&
        magnumBytes[7] == kMagicNumber[7]) {
        return true;
```

```
    }
    return false;
}

void DrwigParser::readMagicNumber() {
    // Read magic number bytes from serial port
    for (int i = 0; i < kDrwigDataMagicNumberSize; ++i) {
        m_serial_object.ReadByte(magnumBytes[i], ms_timeout);
    }
}
```

Thread-Safe Operations

Thread-Safe Queue Implementation

File: `include/ts_queue.hpp`

```
template<typename T>
class TSQueue {
private:
    mutable std::mutex m_mutex;
    std::queue<T> m_queue;
    std::condition_variable m_condition;

public:
    TSQueue() = default;
    TSQueue(const TSQueue& other) {
        std::lock_guard<std::mutex> lock(other.m_mutex);
        m_queue = other.m_queue;
    }

    void push(const T& data) {
        std::lock_guard<std::mutex> lock(m_mutex);
        m_queue.push(data);
        m_condition.notify_one();
    }

    bool pop(T& result) {
        std::unique_lock<std::mutex> lock(m_mutex);
        while (m_queue.empty()) {
            m_condition.wait(lock);
        }
        result = m_queue.front();
        m_queue.pop();
        return true;
    }

    bool try_pop(T& result) {
        std::lock_guard<std::mutex> lock(m_mutex);
        if (m_queue.empty()) {
            return false;
```

```
        }
        result = m_queue.front();
        m_queue.pop();
        return true;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m_mutex);
        return m_queue.empty();
    }

    size_t size() const {
        std::lock_guard<std::mutex> lock(m_mutex);
        return m_queue.size();
    }

    void clear() {
        std::lock_guard<std::mutex> lock(m_mutex);
        std::queue<T> empty_queue;
        m_queue.swap(empty_queue);
    }
};
```

Data Processing Thread

```
void DrwigParser::ParseDataImpl() {
    std::cout << "Data parsing thread started" << std::endl;

    while (read_thread) {
        uint32_t buffer_index;

        // Wait for data from reader thread
        if (!reader_queue.try_pop(buffer_index)) {
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
            continue;
        }

        try {
            // Process the frame
            {
                std::lock_guard<std::mutex> lock(buffer_m_mutex);
                ParseFrame(m_buffer[buffer_index]);
            }

        } catch (const std::exception& e) {
            std::cerr << "Frame parsing error: " << e.what() << std::endl;
        }
    }

    std::cout << "Data parsing thread stopped" << std::endl;
}
```

Data Conversion

Fixed-Point to Floating-Point Conversion

The DRWIG protocol uses fixed-point arithmetic for coordinates and velocities. The conversion factors depend on the specific radar configuration:

```
// Conversion constants (example values - adjust based on radar
configuration)
constexpr float POSITION_SCALE = 0.01f;     // 1 cm resolution
constexpr float VELOCITY_SCALE = 0.1f;      // 0.1 m/s resolution
constexpr float SIZE_SCALE = 0.01f;         // 1 cm resolution

// Convert DRWIG detected object to visualization format
DetectedObject convertDetectedObject(const DrwigDetectedObject& drwig_obj)
{
    DetectedObject obj;

    // Convert fixed-point coordinates to floating-point meters
    obj.position.x = static_cast<float>(drwig_obj.x) * POSITION_SCALE;
    obj.position.y = static_cast<float>(drwig_obj.y) * POSITION_SCALE;
    obj.position.z = static_cast<float>(drwig_obj.z) * POSITION_SCALE;

    // Convert velocity
    obj.doppler_velocity = static_cast<float>(drwig_obj.doppler_velocity)
* VELOCITY_SCALE;

    // Convert signal strength
    obj.peak_value = static_cast<float>(drwig_obj.peak_value);

    // Calculate polar coordinates
    obj.calculatePolar();

    return obj;
}

// Convert DRWIG cluster object to visualization format
ClusterObject convertClusterObject(const DrwigClusterObject& drwig_obj) {
    ClusterObject obj;

    obj.center.x = static_cast<float>(drwig_obj.x) * POSITION_SCALE;
    obj.center.y = static_cast<float>(drwig_obj.y) * POSITION_SCALE;
    obj.center.z = 0.0f; // Clusters are typically 2D

    obj.width = static_cast<float>(drwig_obj.width) * SIZE_SCALE;
    obj.length = static_cast<float>(drwig_obj.length) * SIZE_SCALE;
    obj.height = 1.0f; // Default height

    obj.detection_count = 1; // Unknown from DRWIG data
    obj.confidence = 0.8f;   // Default confidence
```

```
    return obj;
}

// Convert DRWIG tracked object to visualization format
TrackedObject convertTrackedObject(const DrwigTrackedObject& drwig_obj,
uint32_t track_id) {
    TrackedObject obj;

    obj.id = track_id;
    obj.position.x = static_cast<float>(drwig_obj.x) * POSITION_SCALE;
    obj.position.y = static_cast<float>(drwig_obj.y) * POSITION_SCALE;
    obj.position.z = 0.0f;

    obj.velocity.x = static_cast<float>(drwig_obj.vx) * VELOCITY_SCALE;
    obj.velocity.y = static_cast<float>(drwig_obj.vy) * VELOCITY_SCALE;
    obj.velocity.z = 0.0f;

    obj.width = static_cast<float>(drwig_obj.width) * SIZE_SCALE;
    obj.length = static_cast<float>(drwig_obj.length) * SIZE_SCALE;
    obj.height = 1.5f; // Default vehicle height

    obj.confidence = 0.9f;
    obj.age = 1;
    obj.missed_detections = 0;

    // Add current position to history
    obj.addToHistory(obj.position);

    return obj;
}
```

Error Handling

Connection Error Recovery

```
class DrwigParser {
private:
    std::atomic<bool> m_connection_active{false};
    std::atomic<int> m_reconnect_attempts{0};
    static constexpr int MAX_RECONNECT_ATTEMPTS = 5;
    static constexpr int RECONNECT_DELAY_MS = 2000;

    void handleConnectionError() {
        m_connection_active = false;

        if (m_reconnect_attempts < MAX_RECONNECT_ATTEMPTS) {
            std::cerr << "Connection lost. Attempting to reconnect... ("
                      << (m_reconnect_attempts + 1) << "/" <<
MAX_RECONNECT_ATTEMPTS << ")" << std::endl;
```

```
std::this_thread::sleep_for(std::chrono::milliseconds(RECONNECT_DELAY_MS))
;

            try {
                m_serial_object.Close();
                OpenComPort();
                m_connection_active = true;
                m_reconnect_attempts = 0;
                std::cout << "Reconnection successful" << std::endl;

            } catch (const std::exception& e) {
                std::cerr << "Reconnection failed: " << e.what() <<
std::endl;
                m_reconnect_attempts++;
            }
        } else {
            std::cerr << "Maximum reconnection attempts reached. Stopping
parser." << std::endl;
            read_thread = false;
        }
    }
};
```

Data Validation

```
bool DrwigParser::validateFrame(const Frame_t& frame) {
    // Check minimum frame size
    if (frame.size() < kDrwigDataMagicNumberSize +
sizeof(DrwigDataHeader)) {
        return false;
    }

    // Validate magic number
    if (!std::equal(frame.begin(), frame.begin() +
kDrwigDataMagicNumberSize,
                    kMagicNumber.begin())) {
        return false;
    }

    // Extract and validate header
    DrwigDataHeader header;
    std::memcpy(&header, frame.data() + kDrwigDataMagicNumberSize,
sizeof(header));

    // Check header fields
    if (header.total_packet_len > max_frame_size ||
        header.total_packet_len < sizeof(DrwigDataHeader) ||
        header.num_tlvs > 10) { // Reasonable limit
        return false;
    }
```

```
    // Validate frame length matches header
    if (frame.size() != kDrwigDataMagicNumberSize +
header.total_packet_len) {
        return false;
    }

    return true;
}
```

Performance Optimization

Buffer Management

```
class DrwigParser {
private:
    // Pre-allocated buffers for zero-copy operations
    static constexpr size_t BUFFER_COUNT = 4;
    static constexpr size_t BUFFER_SIZE = 32768;

    std::array<Frame_t, BUFFER_COUNT> m_buffers;
    std::atomic<size_t> m_write_buffer_index{0};
    std::atomic<size_t> m_read_buffer_index{0};

    void initializeBuffers() {
        for (auto& buffer : m_buffers) {
            buffer.reserve(BUFFER_SIZE);
        }
    }

    Frame_t& getWriteBuffer() {
        return m_buffers[m_write_buffer_index.load()];
    }

    Frame_t& getReadBuffer() {
        return m_buffers[m_read_buffer_index.load()];
    }

    void switchWriteBuffer() {
        m_write_buffer_index = (m_write_buffer_index + 1) % BUFFER_COUNT;
    }

    void switchReadBuffer() {
        m_read_buffer_index = (m_read_buffer_index + 1) % BUFFER_COUNT;
    }
};
```

Memory Pool for Object Allocation

```
template<typename T>
class ObjectPool {
private:
    std::stack<std::unique_ptr<T>> m_pool;
    std::mutex m_mutex;

public:
    std::unique_ptr<T> acquire() {
        std::lock_guard<std::mutex> lock(m_mutex);
        if (m_pool.empty()) {
            return std::make_unique<T>();
        }

        auto obj = std::move(m_pool.top());
        m_pool.pop();
        return obj;
    }

    void release(std::unique_ptr<T> obj) {
        if (obj) {
            std::lock_guard<std::mutex> lock(m_mutex);
            m_pool.push(std::move(obj));
        }
    }
};

// Usage in parser
ObjectPool<std::vector<DrwigDetectedObject>> detected_object_pool;
ObjectPool<std::vector<DrwigClusterObject>> cluster_object_pool;
ObjectPool<std::vector<DrwigTrackedObject>> tracked_object_pool;
```

Integration Examples

Basic Integration with Visualizer

```
// In RadarVisualizer class
bool RadarVisualizer::initializeParser() {
    try {
        // Create parser instance
        std::string serial_port = m_parserConfig.serialPort;
        m_parser = std::make_unique<DrwigParser>(serial_port,
m_parserConfig.baudRate);

        // Register callbacks
        m_parser->RegisterObjectHandler(
            [this](const DrwigObjectHeader& header, const
std::vector<DrwigDetectedObject>& objects) {
                this->onDetectedObjects(header, objects);
            }
        );
```

```
        m_parser->RegisterObjectHandler(
            [this](const DrwigObjectHeader& header, const
std::vector<DrwigClusterObject>& objects) {
                this->onClusterObjects(header, objects);
            }
        );

        m_parser->RegisterObjectHandler(
            [this](const DrwigObjectHeader& header, const
std::vector<DrwigTrackedObject>& objects) {
                this->onTrackedObjects(header, objects);
            }
        );

        // Initialize and start parser
        m_parser->Init();

        // Start parser thread
        m_parserThread = std::thread([this]() {
            m_parser->Parse();
        });

        m_parserConfig.isConnected = true;
        m_parserConfig.connectionStatus = "Connected";

        return true;

    } catch (const std::exception& e) {
        std::cerr << "Failed to initialize DRWIG parser: " << e.what() <<
std::endl;
        m_parserConfig.connectionStatus = "Connection Failed: " +
std::string(e.what());
        return false;
    }
}
```

Callback Implementation

```
void RadarVisualizer::onDetectedObjects(const DrwigObjectHeader& header,
                                        const
std::vector<DrwigDetectedObject>& objects) {
    std::lock_guard<std::mutex> lock(m_frameMutex);

    // Store for frame conversion
    m_detectedHeader = header;
    m_detectedObjects = objects;

    // Check if we have all object types for complete frame
    if (hasCompleteFrame()) {
        convertToRadarFrame(m_detectedHeader, m_detectedObjects,
```

```
                              m_clusterHeader, m_clusterObjects,
                              m_trackedHeader, m_trackedObjects);
    }
}

void RadarVisualizer::convertToRadarFrame(const DrwigObjectHeader&
detectedHeader,
                                          const
std::vector<DrwigDetectedObject>& detected,
                                          const DrwigObjectHeader&
clusterHeader,
                                          const
std::vector<DrwigClusterObject>& clusters,
                                          const DrwigObjectHeader&
trackedHeader,
                                          const
std::vector<DrwigTrackedObject>& tracked) {

    auto frame = std::make_shared<RadarFrame>();
    frame->frame_number = getCurrentFrameNumber();
    frame->timestamp = std::chrono::high_resolution_clock::now();

    // Convert detected objects
    frame->detected_objects.reserve(detected.size());
    for (const auto& obj : detected) {
        frame->detected_objects.push_back(convertDetectedObject(obj));
    }

    // Convert cluster objects
    frame->cluster_objects.reserve(clusters.size());
    for (const auto& obj : clusters) {
        frame->cluster_objects.push_back(convertClusterObject(obj));
    }

    // Convert tracked objects
    frame->tracked_objects.reserve(tracked.size());
    for (size_t i = 0; i < tracked.size(); ++i) {
        uint32_t track_id = generateTrackId(tracked[i]);
        frame->tracked_objects.push_back(convertTrackedObject(tracked[i],
track_id));
    }

    // Update statistics
    frame->stats.total_detections = static_cast<uint32_t>(
        frame->detected_objects.size() +
        frame->cluster_objects.size() +
        frame->tracked_objects.size()
    );

    // Update current frame
    updateFrame(frame);
    m_newFrameAvailable = true;
}
```

Configuration Panel Integration

```
void RadarVisualizer::renderUARTConfigPanel() {
    if (ImGui::CollapsingHeader("UART Configuration")) {
        // Serial port selection
        ImGui::InputText("Serial Port", m_parserConfig.serialPort,
                         sizeof(m_parserConfig.serialPort));

        // Baud rate selection
        const int baud_rates[] = {115200, 230400, 460800, 921600,
1000000};
        const char* baud_names[] = {"115200", "230400", "460800",
"921600", "1000000"};
        int current_baud = 0;
        for (int i = 0; i < 5; ++i) {
            if (baud_rates[i] == m_parserConfig.baudRate) {
                current_baud = i;
                break;
            }
        }

        if (ImGui::Combo("Baud Rate", &current_baud, baud_names, 5)) {
            m_parserConfig.baudRate = baud_rates[current_baud];
        }

        // Connection controls
        if (m_parserConfig.isConnected) {
            if (ImGui::Button("Disconnect")) {
                shutdownParser();
            }
            ImGui::SameLine();
            ImGui::TextColored(ImVec4(0, 1, 0, 1), "Connected");
        } else {
            if (ImGui::Button("Connect")) {
                initializeParser();
            }
            ImGui::SameLine();
            ImGui::TextColored(ImVec4(1, 0, 0, 1), "Disconnected");
        }

        // Connection status
        ImGui::Text("Status: %s",
m_parserConfig.connectionStatus.c_str());

        // Auto-connect option
        ImGui::Checkbox("Auto-connect on startup",
&m_parserConfig.autoConnect);
    }
}
```

Next: Recording and Playback System | **Previous**: Core Visualizer Module