Mostly parsed by Google Gemini 1.5 Pro

# Son9.pdf

**Key SON Library Components:**

1. `SON.H`**:** This is the main header file you'll include in your C++ code. It declares the necessary functions and data structures.
2. `SON32.DLL` **(and** `SON32.LIB` **if dynamically linking):** This is the library itself, containing the compiled code for the SON functions. Dynamic linking is generally preferred.
3. `son9.pdf`**:** The documentation will be your essential guide for understanding how to use the library functions correctly.

**Core SON Functions You'll Likely Use:**

- **File Creation and Setup:**
  - `SONCreateFile()` or `SONCreateFileEx()`: To create a new SON file to store the converted data. Consider using `SONCreateFileEx()` to enable larger file sizes if necessary.
  - `SONSetFileClock()`: Crucial for time alignment. You'll set `usPerTime` and `timePerADC` to match the 1401's timebase. This ensures that your CANopen data is on the same time scale as your other experimental data.
  - `SONSetRealChan()`: Define a real-valued channel to store the position data from your motors.
- **Data Writing:**
  - `SONWriteRealBlock()`: Use this function to write the motor position data into the SON file. Be sure to provide the correct timestamp for each data block, again based on the synchronized timebase.
- **File Management:**
  - `SONCloseFile()`: When you're finished recording, close the SON file properly.
  - `SONCommitFile()`: Ensures all buffered data is written to disk (important for real-time systems where buffering is used).

**C++ Implementation Outline:**

1. **CANopen to Serial Conversion:** Handle the communication with your CANUSB adapter to receive the motor position data in a serial format that your C++ program can read.
2. **Time Synchronization:** This is the most critical aspect. You'll need a robust mechanism to synchronize the timestamps of the CANopen data with the 1401's clock. Options include:
   - **Hardware Trigger:** If possible, use a hardware trigger signal generated by the 1401 to initiate data acquisition from the CAN bus.
   - **Software Synchronization:** Implement a precise timing scheme in your software using high-resolution timers. Regularly query the 1401's clock and use that time as the basis for the CANopen data timestamps. Be mindful of potential latency and jitter.

3. **Data Conversion and Writing:** Read the serial data from the CANUSB adapter, convert it to the appropriate data type (likely `float` for position), and use `SONWriteRealBlock()` to write it to the SON file with the synchronized timestamp.

4. **Real-time Considerations:** For a real-time system:

    - Use buffering (`SONSetBuffering()`) to avoid blocking data acquisition while writing to disk.
    - Regularly call `SONCommitFile()` to flush the buffers and ensure data is written in a timely manner.
    - Optimize your code for performance to minimize latency and keep up with the data stream.

**Example Code Snippet (Illustrative):**

```
#include "SON.H"
#include <iostream> // For error handling (replace with your logging method)
// ... other includes ...

int main() {
    short fh = SONCreateFileEx("motor_data.son", 1, 0, 1); // 1 channel, big file
enabled
    if (fh < 0) {
        std::cerr << "Error creating SON file: " << fh << std::endl;
        return 1;
    }

    // Set timebase to match 1401 (replace with your actual values)
    SONSetFileClock(fh, 10, 1); // 10 us per time unit, 1 time unit per ADC sample

    // Set up real channel (replace with your channel details)
    int result = SONSetRealChan(fh, 0, 0, 0, 512 * 1024, "Motor Position", "Pos",
1.0, 0.0, "deg");
    if (result < 0) { /* Handle error */ }

    // Main loop (replace with your CAN communication and timing code)
    while (/* acquiring data */) {
        // ... get motor position and synchronized timestamp ...
        float positionData[1024];  // Example buffer size

        // ...populate positionData...
        TSTime timestamp = /* Get synchronized timestamp from 1401 */;

        result = SONWriteRealBlock(fh, 0, positionData, 1024, timestamp);
        if (result < 0) { /* Handle error */ }

        SONCommitFile(fh, 0); // Flush buffers (adjust frequency as needed)

    }

    SONCloseFile(fh);
    return 0;
}
```

Remember to consult the `son9.pdf` documentation for details on each function and adjust the parameters accordingly. This is a simplified illustration; real-world implementation will involve more complex error handling, timing synchronization, and CAN communication code. Test thoroughly and incrementally.