BME68X Sensor Firmware – Developer Documentation

This document explains the structure and functionality of the firmware written in C++ for managing multiple BME68X sensors. The project uses both a main header/source implementation and an external communication multiplexer library (commMux) obtained from BSEC2 example files.

Overview

The firmware controls a set of eight BME68X sensors using SPI communication combined with an I²C-based multiplexer. It supports dynamic configuration via an SD card and can fall back to hardcoded configurations if needed. In addition, it includes functionality for:

- Reading sensor data.
- Configuring heater profiles and duty cycle profiles.
- Cycling through different heater profile assignments using button presses.
- Handling serial commands (such as starting/stopping data collection, uploading configuration, and reporting sensor status).

main.h – Header File

The header file defines:

- **Constants & Definitions:**
  Sensor count, field counts, pin numbers (for buttons, LEDs, and SD card), command strings, and timing constants.

- **Data Structures:**

  - `HeaterConfig`: Wraps a native heater configuration.
  - `DutyCycleProfile` & `DutyCycleState`: Manage duty cycle settings.
  - `SensorConfig`: Holds sensor configuration data loaded from JSON.

- **Global Declarations:**
  External variables for heater configurations, duty cycles, sensor objects, and button states.

- **Function Prototypes:**
  Prototypes for error handling (blinking LEDs, reporting sensor status), heater profile functions (setting and retrieving profiles), configuration loading (from SD card), sensor assignment (dynamic or hardcoded), serial command processing, button handling, data collection, and duty cycle updates.

main.cpp – Source File

The main implementation file includes:

Global Variables Initialization

- Arrays and variables for sensor objects, heater profiles, duty cycle profiles, sensor configurations, and various control flags.
- Hardcoded mappings and a table for cycling through heater profiles.

Error Handling Functions

- **`getBmeErrorMessage()`** returns a string description for error codes.
- **LED Blink Functions:**
  `blinkWarningLED()` and `blinkErrorLED()` indicate warnings and errors using the built-in LED.

Heater Profile Functions

- **`setHeaterProfile()`**: Wraps the native call to configure a sensor's heater profile.
- **`getHeaterProfiles()`**: Retrieves and prints the current heater and duty cycle profiles for all sensors.

Configuration Loading

- **`loadDynamicConfig()`**: Attempts to load configuration data from the SD card (JSON file).
  - If found, it parses heater configurations, duty cycle profiles, and sensor assignments.
  - If not, the firmware uses hardcoded values.
- **Configuration Upload:**
  Functions like `uploadConfigFromSerial()` allow updating configuration via the serial monitor.

Hardcoded Initialization Functions

- **`initializeHeaterConfigs()`**: Sets up fixed heater configurations with predefined temperature and duration arrays.
- **`initializeDutyCycleProfiles()`** and **`initializeSensorDutyCycles()`**: Initialize duty cycle settings for sensors.

Sensor Assignment

- **`assignDynamicSensorConfigs()`**: Assigns sensor configurations based on JSON data.
- **`assignHardcodedSensorConfigs()`**: Uses predefined mappings to assign heater profiles to sensors.

Serial & Button Handling

- **`handleSerialCommands()`**: Reads and processes serial commands such as START, STOP, GETHEAT, etc.
- **`handleButtonPresses()`**: Reads button states (with debounce) to either increment/decrement a label or cycle through heater profiles when both are pressed simultaneously.

Data Collection and Duty Cycle Updates

- **`collectAndOutputData()`**: Reads sensor data, formats it as CSV, and outputs via the serial port.
- **`updateDutyCycleStates()`**: Updates each sensor's duty cycle state based on its profile.

Main Setup and Loop

- `setup()`:
  - Initializes serial communication, multiplexer (via commMux), and sensor objects.
  - Loads configuration from SD card or falls back to hardcoded values.
  - Initializes each sensor and assigns configurations.
- `loop()`:
  Continuously handles serial commands, button presses, and triggers data collection at defined intervals.

External commMux Library

The commMux library is used to configure and manage the communication interface for each sensor. Key points include:

- **Purpose:**
  The library handles the SPI communication via a multiplexer controlled over I²C. This allows one SPI bus to service multiple sensors by selectively activating each sensor's chip select.

- **Key Functions:**

  - `commMuxSetConfig()`: Configures the multiplexer settings for a given sensor index.
  - `commMuxBegin()`: Initializes I²C and SPI interfaces.
  - `commMuxWrite()` and `commMuxRead()`: Handle writing to and reading from sensor registers over SPI after setting the correct multiplexer channel.
  - `commMuxDelay()`: Provides a microsecond delay between communications.

The library is adapted from BSEC2 examples and provides a simple, effective way to share the SPI bus between multiple sensors.

Usage & Extension

- **Configuration:**
  The firmware supports dynamic configuration through an SD card. If configuration files (in JSON format) are available, they will override hardcoded defaults.

- **Command Handling:**
  Serial commands allow control of the sensors (starting/stopping data collection, getting heater/duty profiles, uploading new configuration, etc.).

- **Sensor Operation:**
  Button presses offer manual control. A simultaneous press cycles heater profiles, while individual presses adjust a label counter used in data logging.

- **Extensibility:**
  Developers can add new commands, adjust heater/duty cycle profiles, or extend error handling and logging as needed. The modular structure (separate sections for configuration, sensor assignment, data collection, and serial handling) makes it straightforward to maintain or extend the functionality.

This documentation should serve as a solid overview for developers working with or extending the firmware. For detailed changes or troubleshooting, refer to the inline comments and the corresponding code sections.

Happy coding!