**BMI088 Project  
Document compiled by Rivoo Bagchi.**

Please note that this document, and segments of the code have been updated using generative AI models such as ChatGPT. This driver is originally developed by Parker Hitch (see: [ParkerHitch/STM32-BMI088: A simple STM32 SPI driver for the Bosch BMI088 inertial measurement unit. (github.com)](https://github.com/ParkerHitch/STM32-BMI088)). This document contains details for attempts at updating some features and packaging it to be suitable for the vac-work project – including the implementation of a UART transfer system. Since there are numerous similar projects existing already – albeit with varying degrees of success and deviances to account for different use-cases – this project does not seek to reinvent the wheel from ground-up, instead repurposing public-use code to the same intention.

Please note that this is a preliminary version of the project, subject to significant further changes as only very basic testing has been performed.

1. **BMI088 Executable**
2. **Introduction**The intention of this section of the project is to produce a full executable that is able to be flashed onto the STM32F4 custom PCB. This executable should be able to record and output readings from the BMI088 through the UART protocol (readable on a PC through a COM port.)  
     
   The project has been designed to function using the built-in HAL features available built into the STM32F4 library. Note that it is not necessary to use a #include statement, since the inclusion of the relevant driver has been performed through the project properties. See [OpenSTM32 Community Site | Adding new Include files](https://www.openstm32.org/forumthread3284) for more information.
3. **main.c file**This section details the functions built into main.c. This does not describe the exact process of configuring and reading from the BMI088, but instead focuses on how the data is processed after reading and output using the implementation of UART. This is a brief rundown of all of the functions implemented.

**main()**: The entry point of the program, initializes the hardware peripherals and enters an infinite loop to read accelerometer data and print it over UART. The flag for enabling float formatting is enabled from the project properties to allow this. The reading is performed through a custom function since there are errors in the existing implementation. However, this means that the data is not inherently produced in a readable format. It can be intercepted and interpreted after UART transmission – performed in a later stage of this project.

At this stage, the project is only designed to read accelerometer data. The IMU\_init function can be used with the same argument as Accel\_init, in order to configure for both

// Main loop

**while** (1) {

// Read accelerometer data

**float** rawVals[6];

CUSTOM\_ACCELERATION(rawVals);

// Format the accelerometer data

uint8\_t buffer[100];

**int** length = **snprintf**((**char**\*)buffer, **sizeof**(buffer),

"X=%.2f, Y=%.2f, Z=%.2f\r\n", rawVals[0], rawVals[1], rawVals[2]);

// Transmit the formatted data via UART

HAL\_UART\_Transmit(&huart1, buffer, length, HAL\_MAX\_DELAY);

// Delay for 0.001 second - to be changed

HAL\_Delay(1);

}

}

**SystemClock\_Config()**: Configures the system clock settings – default, unconfigured

**MX\_SPI1\_Init()**: Initializes the SPI1 peripheral with specific configurations, using default configuration at this stage. Can be replaced with SPI1\_Init if IOC configurations are not chosen.

**MX\_USART1\_UART\_Init()**: Initializes the USART1 peripheral for UART communication. Using default configuration at this stage. Can be replaced with a custom UART implementation if necessary at a later stage.

**MX\_GPIO\_Init()**: Initializes the GPIO pins.

**Error\_Handler()**: Handles errors by disabling interrupts and entering an infinite loop – default, unconfigured.

**assert\_failed()**: Reports the file name and line number where an assertion error occurred (used when full assert is enabled) – default

1. **Project Settings**

The project was generated using STM32 CubeMX (hence MX preceding the names of functions) – this effectively provides a simple, HAL-based approach to configuring SPI, UART, and GPIO initialization instead of a bare-metal approach (seemingly unnecessary at this stage as I am not familiar with any specific configurations which require straying from the defaults which STM32 CubeMX provides.) As listed, SPI1 and UART1 are enabled – matching with the pins designated in the PCB schematic. The STM32F4 HAL driver is chosen by default for this project.

1. **Critical information**Note that STMF4### boards use different registers for sending and receiving via UART. More information is available at this link: [Ringbuffer does not work on STM32F4xx boards · Issue #1 · HansPeterHaastrup/STM32Ringbuffer (github.com)](https://github.com/HansPeterHaastrup/STM32Ringbuffer/issues/1).  
   TL;DR – use DR instead of TDR, etc. The functionality is similar, syntax is different.
2. **BMI088 Driver**

Please refer to this github link for the original driver. Minor adjustments have been made in order to make it suitable for this purpose..

**a) Accel.c**

* ***ACCEL\_INIT****(SPI\_HandleTypeDef* spiHandler): Initializes the accelerometer with given SPI handler and reloads settings.
* **ACCEL\_GOOD\_SETTINGS()**: Configures the accelerometer with optimal settings for range, configuration, and FIFO.
* **ACCEL\_SELF\_TEST()**: Performs a self-test on the accelerometer to ensure proper operation.
* **ACCEL\_RELOAD\_SETTINGS()**: Reloads the current accelerometer settings from its registers.
* **ACCEL\_READ\_ID()**: Reads and returns the accelerometer's chip ID.
* **ACCEL\_READ\_ACCELERATION()**: Reads and returns the current acceleration values as a Vector3 struct.
* ***CUSTOM\_ACCELERATION****(float* rawVals): Custom function to read acceleration values into the provided buffer.
* **ACCEL\_READ\_TEMPERATURE()**: Reads and returns the current temperature from the accelerometer.
* **ACCEL\_READ\_SENSORTIME()**: Reads and returns the sensor's timestamp.
* **ACCEL\_READ\_ERROR()**: Reads and returns the accelerometer's error status.
* **ACCEL\_READ\_PWR\_MODE()**: Reads and returns the accelerometer's power mode.
* **ACCEL\_READ\_ACCEL\_ENABLED()**: Reads and returns whether the accelerometer is enabled.
* **ACCEL\_SET\_CONFIG(uint8\_t oversamplingRate, uint8\_t outputDataRate)**: Sets the accelerometer's configuration for oversampling rate and output data rate.
* **ACCEL\_SET\_RANGE(uint8\_t range)**: Sets the accelerometer's measurement range.
* **ACCEL\_WRITE\_PWR\_ACTIVATE()**: Activates the accelerometer's power mode.
* **ACCEL\_WRITE\_PWR\_SUSPEND()**: Suspends the accelerometer's power mode.
* **ACCEL\_WRITE\_ACCEL\_ENABLE()**: Enables the accelerometer.
* **ACCEL\_WRITE\_ACCEL\_DISABLE()**: Disables the accelerometer.
* **ACCEL\_READ\_FIFO\_LEN()**: Reads and returns the length of the FIFO buffer.
* **ACCEL\_READ\_FIFO()**: Reads and parses the FIFO buffer data into an AccelDataBuffer.
* **ACCEL\_WRITE\_FIFO\_ENABLED(uint8\_t enabled)**: Enables or disables the FIFO buffer.
* **ACCEL\_WRITE\_FIFO\_MODE(uint8\_t modeFIFO)**: Sets the FIFO buffer mode.
* **ACCEL\_WRITE\_FIFO\_DOWNSAMP(uint8\_t downsampFIFO)**: Sets the FIFO buffer downsampling rate.
* **select()**: Pulls the chip select line low to initiate communication.
* **unselect()**: Pulls the chip select line high to end communication.
* \**readAddr(uint8\_t addr, uint8\_t* outBuff, int outBytes)\*\*: Reads data from a specified address into the buffer.
* **writeAddr(uint8\_t addr, uint8\_t data)**: Writes data to a specified address.
* ***parseRawUInts****(uint8\_t* rawVals): Parses raw accelerometer data into a Vector3 struct.
* **setRangeMem(uint8\_t range)**: Sets internal memory for the accelerometer's range.

1. **Gyro.c**

* ***GYRO\_INIT****(SPI\_HandleTypeDef spiHandler):*\* Initializes the gyroscope with the specified SPI handler.
* **GYRO\_GOOD\_SETTINGS():** Configures the gyroscope with default settings for range, output data rate, and FIFO mode.
* **GYRO\_READ\_ID():** Reads and returns the gyroscope's chip ID.
* **GYRO\_READ\_RATES():** Reads the current gyroscope rates and returns them as a Vector3 struct.
* **GYRO\_RELOAD\_SETTINGS():** Reloads the gyroscope settings for range and output data rate.
* **GYRO\_SELF\_TEST():** Performs a self-test on the gyroscope and returns the test result.
* **GYRO\_READ\_FIFO():** Reads data from the FIFO buffer and returns it as a GyroDataBuffer struct.
* **GYRO\_SET\_POWERMODE(uint8\_t gyroPowermode):** Sets the power mode of the gyroscope.
* **GYRO\_SET\_RANGE(uint8\_t gyroRange):** Sets the range of the gyroscope.
* **GYRO\_SET\_OUPUT\_DATA\_RATE(uint8\_t gyroODR):** Sets the output data rate of the gyroscope.
* **GYRO\_SET\_FIFO\_MODE(uint8\_t gyroFIFOMode):** Sets the FIFO mode of the gyroscope.
* **select():** Selects the gyroscope by setting the CS pin low.
* **unselect():** Deselects the gyroscope by setting the CS pin high.
* *readAddr(uint8\_t addr, uint8\_t outBuff, int outBytes):*\* Reads data from the specified address.
* **writeAddr(uint8\_t addr, uint8\_t data):** Writes data to the specified address.
* *parseRawUInts(uint8\_t rawVals):*\* Converts raw gyroscope data to radians per second and returns it as a Vector3 struct.

1. **Vectors.c**vSub(Vector3 a, Vector3 b)` function subtracts the corresponding x, y, and z components of vector `b` from vector `a`. It returns a new Vector3 struct representing the result of the subtraction. This function is used for element-wise vector subtraction in various vector arithmetic operations in main.c and IMU.c
2. **IMU.c**

* **SPI1\_Init():** Configures the SPI1 peripheral with specified settings, including mode, direction, data size, clock polarity and phase, NSS mode, baud rate prescaler, first bit, TI mode, and CRC calculation. Can be replaced by implementation of MX\_SPI1\_Init if necessary – depending on settings.
* **IMU\_INIT**(SPI\_HandleTypeDef spiHandle):\* Initializes the accelerometer and gyroscope using the provided SPI handle.
* **IMU\_SETUP\_FOR\_LOGGING():** Sets up the accelerometer and gyroscope with default settings suitable for data logging.
* **IMU\_ENABLE\_ALL():** Enables the accelerometer and gyroscope by activating their power modes and enabling acceleration measurement. It also includes a delay to ensure proper activation.
* **IMU\_READY():** Performs self-tests on the accelerometer and gyroscope and returns a status indicating if both are ready. A return value of 1 means both sensors are ready, while a negative value indicates which sensor failed the self-test (accelerometer or gyroscope or both).