DMU380 SPI interface

# Introduction

This document provides a description of the SPI interface to the DMU380. This document provides a description of the SPI interface to the DMU380. It is a simplified interface as all calibration and configuration will be done via the UART ports.

All SPI interactions are through reading and writing registers. The DMU380 is the slave device so it must wait for the user to query a register and then clock data out.

# Reading the scaled sensor output

For each 3-axis sensor, there is a nibble that controls the sensor and the associated channels. This allows the user to select the sensors and axis that are to be read.

The least significant bit (0x01) in the sensor configuration registers allow for a sensor to be enabled which will cause the data in associated read registers to be set.

Once the sensor is enabled, the output for each channel is enabled to let it be part of a block read of data.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| – | – | – | – | Enable  Z output | Enable  Y output | Enable  X output | Enable sensor |

Thus, if a user set this lower nibble of the rate sensor configuration register to 0x09 (Z enabled and sensor enabled), then when the status register (0x00) is read, the next location output via SPI will be RATE\_Z\_LSB (0x05).

On the other hand, if the rate sensor configuration is set to 0x0F, after the status register is read, the next location will be RATE\_X\_MSB (0x01).

After setting each sensor configuration register, it is suggested that the user read OUTPUT\_BYTES\_EXPECTED (0x4F) to verify how many bytes should be read to obtain all requested sensor data.

By default, each sensor is enabled and all channels are slated for output:

* Rate sensor X, Y, Z (total 6 bytes)
* Accelerometer X, Y, Z (total 6 bytes)
* Magnetometer X, Y, Z (total 6 bytes)
* Temperature sensors (total 4 bytes)

The default value of OUTPUT\_BYTES\_EXPECTED is therefore 24 bytes (6+6+6+4+2 for status).

# Register Map

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **R/W** | **Reg**  **Addr** | **Default** | **Notes** |
| STATUS | R | 0x00  0x01 | 0x00 | Master status |
| RATE\_X | R | 0x02  0x03 | 0x00 | Scaled sensor output, rate sensors output in 7\*pi/2^16  [1260°/2^16] |
| RATE\_Y | R | 0x04  0x05 | 0x00 |  |
| RATE\_Z | R | 0x06  0x07 | 0x00 |  |
| ACCEL\_X | R | 0x08  0x09 | 0x00 | Accelerometer output in 20/2^16 G |
| ACCEL\_Y | R | 0x0A  0x0B | 0x00 |  |
| ACCEL\_Z | R | 0x0C  0x0D | 0x00 |  |
| MAG\_X | R | 0x0E  0x0F | 0x00 | 2/2^16 Gauss |
| MAG\_Y | R | 0x10  0x11 | 0x00 |  |
| MAG\_Z | R | 0x12  0x13 | 0x00 |  |
| BOARD\_TEMP | R | 0x14  0x15 |  | Temperature output is 200/2^16 degrees C |
| RATE\_TEMP | R | 0x16  0x17 |  | Temperature output is 200/2^16 degrees C |
| ACCEL\_TEMP | R | 0x18  0x19 |  | [FUTURE: this does not get output by default] |
| Reserved |  | 0x1A-0x3E |  | [Note: future things can go here, i.e. angles, gps, etc.] |
| CRC | R | 0x3E  0x3F |  | 16 bit CRC-CCITT over the data that is enabled |
|  |  |  |  |  |
| WHO\_AM\_I | R | 0x40 | 0x38 | System identifier |
| FW\_VERSION | R | 0x41  0x42  0x43  0x44 | 0x01  0x00  0x00  <rev> | Firmware version |
| SERIAL\_NUMBER | R | 0x45  0x46  0x47  0x48 |  | Unique serial number |
| OUTPUT\_BYTES\_EXPECTED | R | 0x4F | 0x18 | Total number of bytes expected based on what is configured. |
|  |  |  |  |  |
| Config: Packet rate divider | R/W | 0x50 | 1 | 400Hz (1), 200Hz (2), 100Hz, 50Hz, 25Hz, 20Hz, 10Hz, 5Hz, 4Hz, 2Hz |
| Config: Filter 1 | R/W | 0x52 | ?? | FUTURE: Sets low pass cutoff for rate sensors. FIXME: determine what settings this should be. |
| Config: Filter 2 | R/W | 0x54 | ?? | FUTURE: Sets low pass cutoff for z accel. FIXME: determine what settings this should be. |
| Config: Filter 3 | R/W | 0x56 | ?? | FUTURE: Sets low pass cutoff for x and y accel. FIXME: determine what settings this should be. |
| Config: Orientation | R/W | 0x58 |  | Determine forward, rightward, and downward facing sides |
| Config: X Hard Iron Bias | R/W | 0x5A |  | I2 scaled from [-1,1) |
| Config: Y Hard Iron Bias | R/W | 0x5C |  | I2 scaled from [-1,1) |
| Config: Soft Iron Scale Ratio | R/W | 0x5E |  | U2 scaled from [0,2) |
| Reserved | R/W | 0x60 |  | FUTURE: Heading-Track Offset to use in NAV filter track update mode. |
| Reserved | R/W | 0x62 |  | FUTURE: Sets yaw rate above which tilt feedback is attenuated. |
| Config: Soft Iron Angle | R/W | 0x64 |  | Angle of major axis of the ellipse generated by 360 degree swing in radian |
| Config: Roll offset Alignment | R/W | 0x66 |  | Roll offset from AHRS body frame to reference body frame in radian |
| Config: Pitch offset Alignment | R/W | 0x68 |  | Pitch offset from AHRS body frame to reference body frame in radian |
| Config: Yaw offset Alignment | R/W | 0x6A |  | Yaw offset from AHRS body frame to reference body frame in radian |
|  |  |  |  |  |
| RESET | R/W | 0x70 | 0x00 | Software, sensor, or algorithm reset. |
| RATE\_CONFIG | R/W | 0x71 | 0x0F | Lower nibble is a bitmask for which channels are output; higher nibble is the range (R only). |
| ACCEL\_CONFIG | R/W | 0x72 | 0x0F | Lower nibble is a bitmask for which channels are output; higher nibble is the range (R only). |
| MAG\_CONFIG | R/W | 0x73 | 0x0F | Lower nibble is a bitmask for which channels are output; higher nibble is the range (R only). |
| TEMP\_CONFIG | R/W | 0x74 | 0x07 | Lower nibble enables each temp sensor, output. |
| Reserved (ANGLE\_CONFIG) | R/W | 0x75 | 0x00 | FUTURE |
| Reserved (NAV\_CONFIG) | R/W | 0x76 | 0x00 | FUTURE |
| SYNC MODE |  | 0x77 |  | FUTURE: bitmask: 1kHz sync input from user, 1PPS from user, 1PPS output, 1kHz output, or sample rate output.  (If the unit has GPS, the sync is 1PPS from GPS and this register is not modifiable.)  Another bit: data ready output is reflected in this register. |
| Reserved (FIFO\_CONFIG) | R/W | 0x78 | 0x00 | There is no FIFO initially. |
|  |  |  |  |  |
| ALGO CONTROL | R/W | 0x7C  0x7D |  | Facilitate magnetometer cal and leveling options. Stages and user instruction. (FIXME: This requires more information and may be from BIT status registers from UART.) |
|  |  |  |  |  |
| Reserved (FACTORY MODE) | R/W | 0x7E  0x7F | 0x0000 | 0x1234 to put the unit into serial mode … this will write to the faux-EEPROM to go into UART mode. |

## STATUS

BITstatus as described in the serial specification with a few modifications:

|  |  |  |  |
| --- | --- | --- | --- |
| **BITstatus Field** | **Bits** | **Meaning** | **Category** |
| masterFail | 0 | 0 = normal, 1 = a fatal error has occurred | BIT |
| HardwareError | 1 | 0 = normal, 1= An internal hardware error. | BIT |
| comError | 2 | 0 = normal, 1 = A communication error | BIT |
| softwareError | 3 | 0 = normal, 1 = An internal software error | BIT |
| Reserved | 4:7 | N/A |  |
| masterStatus | 8 | 0 = nominal, 1 = hardware, sensor, com, or software alert | Status |
| hardwareStatus | 9 | 0 = nominal, 1 = error | Status |
| comStatus | 10 | 0 = nominal, 1 = previous data was overwritten by new data before it was read. | Status |
| softwareStatus | 11 | 0 = nominal, 1 = error | Status |
| sensorStatus | 12 | 0 = nominal, 1 = data ready | Status |
| Reserved | 13:15 | N/A |  |

## CRC

The CRC is taken over the enabled data. (See 16-bit CRC-CCITT in DMU Serial document.)

## WHO\_AM\_I

This register identifies the part, always returning 0x38. This value is factory programmed. Consult the factory for custom alternate values.

## FW\_VERSION

This register identifies the firmware version of the part.

## SERIAL\_NUMBER

This register identifies the firmware version of the part.

## OUTPUT\_BYTES\_EXPECTED

Given the sensor configuration, this register reports how many bytes between STATUS and the end of the sensor block should be read contiguously. This is a read only value and should be used to verify configuration is as intended.

## CONFIG

The configuration fields are intended to be similar to those found in section 7.3 of the DMU serial interface.

## RESET

This command will reset all or part of the system.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| 0 | 0 | 0 | 0 | State estimation algorithm reset | Filter reset | Sensor subsystem reset | Software reset |

Setting this register to 0x01 will performs a core CPU reset, functionally equivalent to a power cycle. All default power-up field settings will apply. The bit will reset to zero when the appropriate thing has finished restting.

## RATE\_CONFIG

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| – | – | – | Range | Enable  Z output | Enable  Y output | Enable  X output | Enable sensor |

Bit 0: Enable sensor sets the rate sensor to be on and running. Bit 1-3: Enable X, Y, and Z parameter make the MSB and LSB values available via the data registers. If the enable channel bits are set, the sensor channel will be returned when reading a block of data. If they are not set, a block read will skip over the associated data and move to the next sensor.

Bit 4: If clear, the rate sensor range will be 250 dps. If set, the rate sensor range will be 450dps.

## ACCEL\_CONFIG

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| – | – | Range1 | Range0 | Enable  Z output | Enable  Y output | Enable  X output | Enable sensor |

|  |  |  |
| --- | --- | --- |
| Range 1 | Range 0 | Full scale range |
| 0 | 0 | 2 G |
| 0 | 1 | 4 G |
| 1 | 0 | 8 G |
| 1 | 1 | Reserved |

## MAG\_CONFIG

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| – | – | Range1 | Range0 | Enable  Z output | Enable  Y output | Enable  X output | Enable sensor |

|  |  |  |
| --- | --- | --- |
| Range 1 | Range 0 | Full scale range |
| 0 | 0 | 0.88 Ga |
| 0 | 1 | 1.9 Ga |
| 1 | 0 | 4.0 Ga |
| 1 | 1 | 5.6 Ga |

## TEMP\_CONFIG

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| – | – | – | – | - | Enable gyro temp sensor output | Enable board temp sensor output | Enable board temp sensor |

# Implementation

The goal of this interface is to implement it in such a way that it uses the same global variables as the UART interface. However, because it is interrupt based, the variables need to be protected so that they remain unmodified for the data acquisition and algorithm tasks.

Thus, the user communication task (TaskUserCommunication) should be used to update the SPI interface variables. While this may mean there are multiple copies of data, they will remain self-consistent at any time.

## Testing

Testing the SPI interface is likely to be non-trivial, requiring code on another processor.

Writing the test code would be most efficient on another IAR evaluation board. This allows for a lot of code reuse with the DMU380 code.

This SPI Test code is the master of the SPI bus, controlling the DMU380 will:

1. Implement a command line UART interface (similar/copy of the debug USART)
2. Implement a SPI (master) driver for another evaluation board, this should be very similar to the SPI driver used for the rate sensor.
3. Create a higher level for the DMU380, configuring the unit to output in different ways. Suggested functions:
   1. Init() will initialize the DMU380, checking the WHO\_AM\_I register
   2. PollForSensorReadings(float sensors[12]) will read the status register until it indicates data is ready. Then it will translate the values into normal units (hence the float).
   3. PollForAccelReadings(float sensors[3]) will re-configure the system to make the accelerometers be the output.
   4. WaitForDataReadyInterupt(float sensors[12]) will configure the data ready from the DMU380 as an interrupt input, reading the data via SPI when it is ready (again, this should be very similar to the rate sensor implementation on the DMU380.
4. The command line should choose between these functions and output the results.

Note that this test code is likely to become example code to users so it is important to make it as clean and readable as possible.

Finally, the TI MSP430 is another processor that could be the one that runs the SPI Test code. This processor may be one that clients use and it may be useful to MEMSIC as a translator to other protocols. However, there will be less code and system knowledge reuse; it will probably take longer to implement.

## DMU380 Hardware setup

A calibration flag should be added to indicate the unit is to go from user UART mode (USER\_A\_UART and USER\_B\_UART) to SPI mode. Upon flash clear, the unit should be in user UART mode to allow for calibration.

|  |  |  |
| --- | --- | --- |
| Pin | User UART mode | User SPI mode |
| PC10 | User UART4\_TX | SPI3\_SCK  Input |
| PC11 | User UART4\_RX | SPI3\_MISO  Output |
| PC12 | Aux UART5\_TX | SPI3\_MOSI  Input |
| PD2 | Aux UART5\_RX | (should be configured as an input, no pullup/down) |
| PA15  (connected to PD2) | Should be configured as an input, no pullup/down | SPI3\_SELECT  Input |
| PB9 |  | User data ready  Output |

The SPI lines need to be configured according to the table (see the rate sensor SPI driver for how though remember, that the DMU380 is a master, the rate sensor is a slave; for this interface, the DMU380 will be a slave.)

Please review [Wikipedia’s SPI article](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus) for SPI idiosyncrasies.

It will likely be useful to review the ST example code for creating a SPI slave. Open IAR without a workspace, and then choose Example projects from the main window, ST, STM32F2xx, STM32F2xx stdperiph lib, SPI. Then choose a temporary location (you probably want to look at the code, not keep it so put it somewhere you can delete it easily). This will open a workspace, select the SPI\_DataExchangeInterrupt example project. This configures a slave and a master, assuming two processors are going to talk to each other. The readme.txt describes the code, pay attention to code marked SPI\_SLAVE.

## Slave SPI Driver

The SPI interface should be interrupt based. The first byte received by the DMU380 will be the address. If the highest bit is set, the goal the DMU380 will write the data that comes after. If the highest bit is clear, the DMU380 will read the register, returning the value with the next clocked signal.

On SPI select: gUserSpi.firstByte = TRUE, place 0 into SPI tx buffer

SPI Slave RX interrupt:

If gUserSpi.firstByte

gUserSpi.addr = rx & 7F;

gUserSpi.writeData = (rx & 0x80) > 0;

if (!gUserSpi.writeData) {

put gUserSpi.Registers[gUserSpi.addr] into the tx buffer

increment gUserSpi.addr according to config variables

deassert data ready // any read clears it

}

else // not the first time, continuing read or write

if (gUserSpi.writeData) {

if (gUserSpi.addr) is valid to write to {

gUserSpi.Registers[gUserSpi.addr] = rx buffer

gUserSpi.DataChanged = TRUE;

place 0 into tx buffer

} else {

place FF into tx buffer

}

} else { // reading data

put gUserSpi.Registers[gUserSpi.addr] into the tx buffer

}

increment gUserSpi.addr according to config variables

## Interface code

There should be a UserSpiUpdate() function called from TaskUserCommunication. This looks at the changes in the gUserSpi.Registers and makes them to the gConfiguration registers, resetting parts of the algorithm (or system) as needed.

It also should update the gUserSpi.Registers that are involved with sensor output (i.e. accel, rate and mags).

If data ready is not cleared, write an error to the status register (comm bit). Otherwise, clear the comm bit error, set the data ready error, and assert the data ready line.