AcubeSAT

NMEA & GPS module integration with STM32 & FreeRTOS

AcubeSAT-OBC-BC-002

Dimitris Stoupis

May 29, 2020
Version: 1.1

Aristotle University of Thessaloniki

Aristotle Space and Aeronautics Team
CubeSat Project

2020

Contents

Changelog

Date	Version	Document Status	Comments
29/05/2020	1.1	PUBLISHED	Converted to new format & fixed typos for publication
30/08/2019	1.0	INTERNALLY RELEASED	Initial revision

This is the latest version of this document (1.1) as of May 29, 2020. Newer versions might be available at https://helit.org/mm/docList/AcubeSAT-OBC-BC-002.

Acronyms

API Application Programming Interface

DMA Direct Memory Access Controller

GPS Global Positioning System

IRQ Interrupt Request

MCU MicroController Unit

NMEA National Marine Electronics Association

NVIC Nested Vectored Interrupt Controller

RTC Real Time Clock

TC Transfer Complete

USART Universal Synchronous/Asynchronous Receiver/Transmitter

1 Summary

In CubeSat missions, the Global Positioning System (GPS) plays a key role in determining position and velocity, two much needed parameters that help in the determination of the satellite's trajectory. In this report, the NMEA protocol, a standard used in a broad range of navigation devices, is presented and the general structure of the protocol commands is explained. Furthermore, a detailed explanation of the GPS code developed for our CubeSat is provided, along with some proposed tests. Finally, code improvements independent of the GPS development made to the CubeSat code are presented and some details are provided.

2 NMEA 0183 Protocol

Recomended file: NMEA0183.pdf
Recomended website: http://www.gpsinformation.org/dale/nmea.htm

The NMEA controls the standard protocol of communication for a broad range of navigation devices and marine electronics. The NMEA 0183 protocol, used in this report, uses simple ASCII characters, and the communication is done using serial communication protocol like Universal Synchronous/Asynchronous Receiver/Transmitter (USART). The protocol's data are transmitted as a structured ASCII string called a **sentence** from a **talker** to multiple **listeners**.

2.1 Sentence Structure

All NMEA sentence characters are printable ASCII characters and there are some special characters from the printable set defining the NMEA sentence. Every sentence begins with the dollar sign character "$" and ends with a carriage return followed by a linefeed or "<CR><LF>". There is also the asterisk ("*") special character which denotes that the check-sum of the sentence follows.

Apart from the special characters defining a NMEA sentence, all fields of the sentence are separated by commas and if a field does not have data available it is just left blank with no space. The total length of the sentence cannot exceed **82** characters in total, including the initial character and the linefeed.

An example of a NMEA sentence could be the following:

```
$GPGLL,4916.45,N,12311.12,W,225444,A,*1D[CR][LF]
```

more sentences and additional information about each sentence field can be found at this helpful website.

2.2 Sentence Request Command Structure

Apart from the sentences sent automatically by the talker, which is the navigation device in our case, there are also the command request sentences, where a user can request a specific sentence, always within the range of sentences supported by the device, to be sent through the serial communication channel.

Command request sentences are no exception regarding the standard NMEA sentence structure illustrated in subsection 2.1. There is also a special characteristic for the request commands and that is they have a fixed length equal to **12** characters with starting special character and line feed included.

The general structure of a request command is as follows:

```
$ttllQ,sss[CR][LF]
```

where **(tt)** is the talker identifier of the device requesting the sentence, **(ll)** is the talker identifier of the device accepting the request and **Q** is always fixed in that position indicating sentence query. The last field **(sss)** contains the sentence identifier to be requested.

Further information regarding the talker identifiers, sentence identifiers and much more can be found in NMEA0183.pdf.

3 Complete Tasks

Source code: acubesat/obc/freertos-mockup

The GPS code development is complete, with a fully functional NMEA sentence parser and a sentence request generator. Up to the date of this report the GPS code has not been tested on the actual Mock-Up to have a full overview on the functionality of the code.

3.1 Hardware

The principle remains the same across almost all devices supporting the NMEA 0183 protocol, as explained in section 2, and it is that the communication is done through an USART channel. Using that to our advantage, we can write module and receiver independent code and we can also configure hardware connection independent of any receiver. Following that logic in this report, the standard communication channel used for the GPS device on the MicroController Unit (MCU) is USART3, set at a rate of **9600 baud**. A schematic representation of the connections between the **blue-pill** and the **GPS module** is shown in Figure 1.

In addition to the above connections, every GPS receiver has an output pin that produces a ***very accurate clock pulse***, with an exact period of **1 second** (in some receivers this period is configurable). In this report such a pin is not used, because for the moment it is not needed for any synchronization operations.

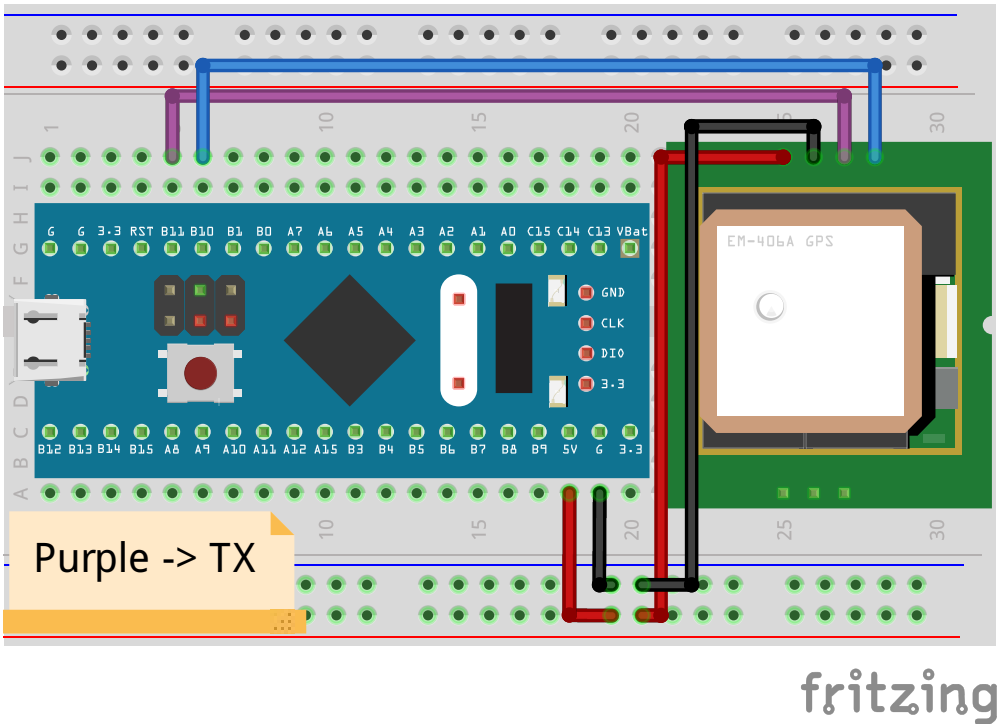

***Figure 1**: GPS Module Connection Diagram (created using fritzing)*

3.2 Communications

The communication channel, as described in section 3, uses the USART3 peripheral of the STM32F103C8 MCU. The USART3 pins are **PB10** for **TX** and **PB11** for **RX**, so the UART **RX** pin of the GPS receiver connects to **PB11** and the **TX** pin of the GPS receiver connects to **PB10** of the MCU.

Also note that all UART transactions on the MCU are accomplished using Direct Memory Access Controller (DMA) channels for the USART3 peripheral, in order to use fewer of the CPU resources. The DMA controller for USART3 specifically is DMA1.

Adding to the above, **idle line** detection interrupt is enabled for the USART3, as it is a crucial part of the sentence reception chain. For more information regarding the interrupt priority and handling refer to subsubsection 3.3.4.

3.2.1 Sentence Reception Implementation

For the reception of NMEA sentences, the MCU is configured so that **channel 3** of the DMA1 controller is always active and waiting for data. The data length for the DMA1 channel 3 buffer is defined in the `GPSTask.c` file as `DMA_RX_BUFFER_SIZE` and it is set to 80 characters, which is equal to `MINMEA_MAX_LENGTH` from the **minmea** library *(check subsubsection 3.3.1 for library details)*.

Whenever a sentence is received and an idle line is detected on USART3, which means

that all characters are transferred, the idle line interrupt is triggered, which initiates message reception from the DMA1 channel 3 buffer. Finally, after successful data transfer, the DMA1 channel 3 is waiting again for the next round of sentences.

3.2.2 Sentence Request Implementation

The sentence request sending operations on USART3 are accomplished using the DMA1 controller on **channel 2**. The data length for the DMA1 channel is set to a fixed character length, equal to **12**, as explained in subsection 2.2. The DMA1 channel 2 is by default disabled and whenever there is a need for data transmission it is enabled. Also the Transfer Complete (TC) interrupt is enabled for this DMA channel, which is triggered when the DMA data have completed transferring through USART, meaning that it is ready for another transmission. After successful transmission the DMA1 channel 2 is disabled to be ready for the next transmission.

The reason that the DMA1 TX channel *(channel 2* is by default disabled, is to enabled other DMA channels to execute their transactions, since only one channel at a time can execute transactions on the DMA controller.

3.3 Software

The latest version of the code can be found on the GPS_Code_Test_Ready tag of the GitLab repository. This code up to the date of this report has only been tested using some dummy sentences sent through USART and all the tests were successful, giving off the desired result.

3.3.1 NMEA Sentence Handling Library

For the parsing and handling operations for NMEA 0183 sentences, the minmea library from the GitHub user Kosma Moczek is used in the code.
A brief explanation of the library usage can be found below:

- First the NMEA sentence is validated through calling the `minmea_sentence_id()` function.
- The value returned by the above operation is passed to a switch statement, which includes cases for `MINMEA_SENTENCE_XXX` defines, where `XXX` is the sentence identifier.
- After a successful case match, the appropriate structure for the sentence, with the name `minmea_sentence_xxx`, is defined and then the valid sentence is parsed using the appropriate parsing function `minmea_parse_xxx()`. The sequence `xxx` denotes the sentence identifier.

3.3.2 GPS Setup

In the GPS setup function denoted as `vSetupGPS()`, the USART3 pins are initialized first and then the USART3 peripheral is initialized. In the initialization of the USART3

peripheral the DMA request register for both RX and TX is enabled and also the idle line detection interrupt is enabled. The interrupt priority of the USART3 Interrupt Request (IRQ) routine, is set to **12** in the code line below:

```
NVIC_SetPriority(USART3_IRQn,
NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 11 + 1, 0));
```

The priority **12** is the highest possible priority that an interrupt can have in order to able to call FreeRTOS Application Programming Interface (API) functions within the interrupt handler. An important remark to make here is that if you want to use any FreeRTOS API function within the interrupt, then the interrupt's priority has to be numerically at or above **12**. Some additional details to keep in mind are:

- Interrupt priorities, using NVIC, have to be set above the `configMAX_SYSCALL_INTERRUPT_PRIORITY`, whose value is set to **decimal 11** in our case, if FreeRTOS API calls are done in the interrupt handler. If no FreeRTOS API calls are executed in the interrupt handler, then the interrupt priority can be set below `configMAX_SYSCALL_INTERRUPT_PRIORITY`, but above zero.
- Whenever a FreeRTOS API call is executed in an interrupt handler, the ISR equivalent function has to be used, otherwise we are looking for trouble!
- Specifically for the TaskNotify function, we have to create a `TaskHandler_t` variable and store the task handler for the desired task to be used. We indicate that explicitly, because due to a misunderstanding we were using the task's name, which is obviously wrong. So be careful with the parameters of the FreeRTOS functions.

After USART3 initialization, the DMA1 RX and TX, channel 3 and 2 respectively are initialized, with TC interrupt enabled on channel 2. Then again the same rules, about the priority setting, apply for the DMA Nested Vectored Interrupt Controller (NVIC), since FreeRTOS API function calls are made.

Finally before finishing the setup, the `xGPSQueue` is created.

3.3.3 GPS Task

In general, the purpose of the GPS task is to wait for message reception from the GPS message queue defined as `xGPSQueue`. Data is appended to the queue using the `osQueueGPSMessage(const char * format, ...)` function. Once a message is available in the queue the GPS task starts to work on that message. Firstly it tokenizes the queue message string using `[CR][LF]` as a delimiter and loops over the string tokens, if there are any left. Then there is a call to the `cGetGPSData` function which validates the received message and if valid saves the appropriate data to the `xGPSData` structure.

The final GPS task will not differ much up to the level described above. However the current GPS task just sends the decoded NMEA sentence data through USART1, so that we know it works. The final version of the task will definitely do more, like notifying other tasks of data reception, updating Real Time Clock (RTC) and **logging** any errors

that occur during sentence handling .

3.3.4 Message Reading Task

The `vGPSMessageRXTask(void *pvParameters)` is a small and elegant task, but with a very important function and that is to read the sentence data from the DMA buffer, when a new sentence has arrived. The task is notified using a FreeRTOS `TaskNotifier`, from the idle line interrupt of the USART3, in which the `DMA_GPS_RX_ISR()` is called, counting the notifier. Once the notification is received the task then calls the private function `prvGPSDMAMessageRX(char *pcRxMessage)`, which takes care of the DMA operations and reads the data from the buffer.

Regarding the `prvGPSDMAMessageRX(char *pcRxMessage)` function, an important note to make here is that any DMA channel has to **disabled** first in order to set the buffer size and also some other registers. The function does that and re-enables the channel after setting the required data.

The last action performed by the `vGPSMessageRXTask(void *pvParameters)`, is to put the received message to the GPS message queue for processing. This is done in the statement `osQueueGPSMessage("%s", cRXMessage);`.

3.3.5 Command Request Sending

Currently the command request `vRequestGPSData(int8_t cNmeaCommand)` function is not used in the code, but it has been tested and it works fine. The purpose of this function is so that the user provides any of the `MINMEA_SENTENCE_XXX` defines, and a request command for the desired sentence is generated and sent over to the GPS receiver through the communication channel.

After successful generation of the command string, the string is then passed as the argument to the private function `prvGPSDMAMessageTX(char *pcTxMessage);` which handles the sending of the command message through USART3 using the DMA channel 2. When the transfer is complete, the DMA channel's interrupt is triggered and the DMA1 channel 2 is disabled, waiting for the next transaction.

4 General code improvements

Outlined below are some code improvements made, which are not related to the GPS development.

4.1 Using Interrupts for USART1 TX Operations

Before the improvement outlined in this report `vUARTTask(void *pvParameters)` task had a while loop, blocking it's operation until the TC flag of the corresponding DMA channel was set. In that way valuable CPU ticks were used and valuable time was wasted in the task, especially if the strings were a bit long.

Now the TC interrupt of the **DMA1 channel 4** is used. After the DMA channel for the USART1 TX is correctly set in the task's code and the channel is enabled, the task then waits until it gets notified by the `TaskNotifier` used for this purpose. The notification is given from the transfer complete IRQ of the DMA1 channel 4. The rules for the NVIC priority, outlined in subsubsection 3.3.3, apply for the DMA1 channel 4 interrupt too, because FreeRTOS API function calls are made.

4.2 Making Pointers Safer

We initialized to NULL some pointers when they were free of an address to avoid nasty behaviors if a pointer is used uninitialized.

5 Bugs Encountered

5.1 The Million Dollar Bug

The main and most time consuming bug encountered in the development of the GPS code, is the priority setting for the interrupts. At first the priorities were set to a relatively random number and of course when a call to a FreeRTOS API function was made, the whole software was crashing. After careful investigation, running the code with the debugger line by line in the interrupts, getting inside some functions line by line and some bit of FreeRTOS manual reading, the bug was gloriously found.

5.2 Not Enough Memory

A general quote is that we have to start to consider using another MCU with a greater RAM capacity or use some external RAM extensions, because when all tasks were on, a *"Not enough memory"* error assertion was made. This was solved by reducing the size allocated for the `xGPSQueue`.

5.3 Not Enough Memory Error After Some Time

After some several successful transactions of sentences through the USART3, a *"Not enough memory"* error was asserted and the reason was a forgotten pointer memory freeing in the `vGPSTask`, namely the `xSentence` variable. So please careful with your pointers!

5.4 NMEA Sentence Parsing Error

When the USART3 rate was set to 115200 baud, 6 out of 10 times there was a NMEA sentence parsing error, despite sending a correct sentence. This bug was solved by reducing the rate to 9600 baud, the same as most GPS receivers.

I am speculating that this bug exists in high speeds because of the way the current code is implemented. What I mean is that probably when the UART rate is sufficiently high, the data from the DMA RX buffer are read before all data are actually transferred into the buffer, resulting in incomplete sentence reading. If high speeds are to be used in the final design, we definitely need to consider this problem.

6 Incomplete Tasks

There are definitely some important incomplete tasks and those tasks among with others are listed below:

- Code test pending
- Sentence and general error logging
- Use GPS received parameters instead of just outputting through USART1
- Sync RTC with the received GPS time
- Use the clock signal provided from the GPS receiver for synchronization, if needed
- Find a GPS receiver that can actually be used in the CubeSat

7 Conclusion

The GPS code is independent of any particular receiver or module and it can work with any device supporting the NMEA 0183 protocol. As outlined throughout the report, an important note for the GPS code operation is the interrupt priorities set for the USART and DMA channel in use for the GPS. Because all interrupts call at least one FreeRTOS API function, their priorites have to be set to at or numerically above **12**. This is dependant to the MCU used so in order to be sure about the numerically lowest value allowed, check the `FreeRTOSConfig.h` and more specifically the `configMAX_SYSCALL_INTERRUPT_PRIORITY` define.