My Journey Understanding the u-blox M10 Global Navigational Satelite System Interface Description Version 1.0.1

By R. G. Sparber

Scope

The u-blox M10 Global Navigational Satelite System (GNSS) receiver has a 176-page Interface Description. It was a lot for me to comprehend, and I was successful only after assistance from the Sparkfun forum.

My goal is to point out major stumbling blocks I encountered while trying to send the M10 a command. Once you understand these problems, you should be able to send the M10 any command.

I will limit my discussion to UART control of the M10. The Interface Description mentions an I2C interface but never provides details. The Sparkfun driver contains this information if you wish to dig it out of the code. Fortunately, this code is well-written and has clear variable names. Thanks, PaulZC (aka Paul Clark).

If you just want to control the M10, the Sparkfun drivers are your answer. I wanted to understand the Interface Description so I could control the M10 myself.

PaulCZ of the Sparkfun forum gave me code that turns off all NEMA sentences. This mae it made it easier to see the acknowledgments from my commands. His code is beyond the scope of this article.

Conclusion

Given the correct sequence of 17 bytes, I can change the state of the M10, so it provides positional data up to an altitude of 50,000 meters. Without this change, I only get the time above 12,000 meters.

In my humble opinion, the u-blox neo M10 Interface Description is poorly written. I present many examples in the following pages. If they had just included an actual array of bytes, it would have been far easier to follow.

Contents

Scope 1
Conclusion 2
Key Reference 2
Change Record 2
High-Level View 3
Medium Level View 3
Low Level View 4
u-blox neo M6 15
Acknowledgments 15

Key Reference

You can find the manufacturer's M10 Interface Description here. All page numbers I mention point to this document.

Change Record

1.0.0 original document
1.0.1 conclusion revised; fixed length LSB/MSB error corrected, updated page 15 to identify ACK-ACK, corrected my confusion about where little-endian is used.

High-Level View

The state of the M10 is defined in its RAM. Commands enable us to clear, set, and read this RAM[2]. Each piece of data in the RAM has a name and associated number.

Medium Level View

The central command for setting a state in the M10 is UBX-CFG-VALSET which lets me set a value. It is part of the UBX-CFG class explained in section 3.10.

The address of each value is called the Key ID. Section 4.8 Configurtion overview has a list of Groups where each group is a related collection of Configuration items. A Configuration item is text that corresponds to a Key ID. For example, CFG-NAVSPG-DYNMODEL has a Key ID of 0x20110021.

Don't get confused here. VALSET has a Class *and* ID, while CFG-NAVSPG-DYNMODEL has a Key ID.

All valid commands cause the M10 to return an acknowledgment called ACK-ACK. If the command is valid but some parameters are not, you will get an ACK-NAK. If the command is not valid, no ACK is returned. For example, if the checksum is wrong, you will not get an ACK.

[2] Data in RAM comes from battery-backed-up RAM (BBR), Flash, and ROM. Any data in BBR takes precedence over Flash and ROM. Flash takes precedence over ROM. BBR's battery lasts about 5 hours so it is useful for warm starts of the GNSS. Flash is external to the M10 and not available on the Sparkfun board. ROM holds the factory settings.

Low Level View

At this level, we look at the details of messages.

Of special note is that the UBX frame starts with a two-character preamble, with the first byte equal to 0xb5 and the second byte 0x62.

The preamble is followed by the class and ID, which identify the message. There are many commands within a class and a few message IDs within each message.

Next is the length, in bytes, of the payload. This is where things start to get a little tricky. It is hard to imagine a need for more than 255 bytes of payload, yet the design provides two bytes. Furthermore, the length is written as "little-endian," which means that the least significant byte comes first.

For example, Say we want to use the message with a class of 0x06, a message ID of 0x8a, and a payload of 9 bytes long:

0xb5 0x62	is the preamble (aka Header)
0x06	is the Class
0x8a	is the message ID
0x09	the least significant byte of the Length
0x00	the most significant byte of the Length

So we would send the M10, in HEX:

B5 62 06 8a 09 00

Although these bytes are not numbered, B5 is byte 0 and it is sent first so the sequence of bytes is sent in little-endian order.

Next, we would send the payload. To understand the payload, we must check back in with the Table Of Contents. There, you will find section 3.8 UBX messages overview.

Every valid message sent to the M10 causes the M10 to return an acknowledgment. This ACK can be a success, ACK-ACK, or a failure, ACK-NAK. If no ACK is returned, the message is not valid. For example, if the payload length is wrong, you might not get an ACK.

The next category of commands relates to the memory that defines the state of the M10. During operation, the M10 reads from its RAM. This memory is initialized using data from ROM, optionally, EEPROM, and battery-backed-up RAM. These are described in section 4.3 Configuration layers.

I will focus only on RAM here because I plan to set the M10's state when I start my program, which will be shortly after my system receives power.

With the Configuration and command messages we have a few types.

We can command the reinitialization of memory from EEPROM, battery-backed-up RAM, and ROM, or the clearing of all data except that stored in ROM.

The next type relates to the setting of individual values. We can delete values and set them. The payload holds a list of which values we want to address and what value we want them to be.

We can also get values, which is called a poll request.

The remaining types of messages are beyond the scope of my document.

Starting at 3.9 UBX-ACK (0x05), we get into the details of each message.

3.9.1 UBX-ACK-ACK (0x05 0x01) and 3.9.2 UBX-ACK-NAK (0x05 0x00) and fundamental to controlling the M10 so we will dig into them. However, they will make more sense after we study a message.

3.10.5 UBX-CFG-VALSET (0x06 0x8a) is the centerpiece of my journey. It enables me to set a value in RAM.

Page 41 of the Interface Document refers to the first two bytes of the message as the *preamble*. That name appears to change to *header* when we reach the details of UBX messages. Yes, it is a minor point, but when I was trying to understand what to do, it was a stumbling block.

I also was frustrated by 3.7 UBX message example. Their example isn't an example at all! With all data symbolic, there was no way to compare actual values.

My big breakthrough came when clive1 on the Sparkfun forum gave me the entire sequence of bytes:

```
uint8_t ubx_cfg_valset_dyn6[] = { // Series 9 and 10
0xB5,0x62,0x06,0x8A,0x09,0x00, // Header/Command/Size  UBX-CFG-VALSET (RAM)
0x00,0x01,0x00,0x00,0x21,0x00,0x11,0x20,0x06, // Payload data (0x20110021 CFG-NAVSPG-DYNMODEL = 6)

0xF2,0x4F }; //Fletcher checksum
```

With this "Rosetta Stone," I was able first to test it and confirm that the M10 returned the correct ACK-ACK. Then, I dissected the array to understand where my thinking was wrong.

The array `ubx_cfg_valset_dyn6[]` is sent via `Serial1` as a series of bytes using the `Serial1.write()` command. More on this later.
Let's take this apart as we follow along in the Interface Description.

`0xB5`	HEADER for UBX-CFG
`0x62`	HEADER for UBX-CFG
`0x06`	class for UBX-CFG-VAL
`0x8A`	ID for SET

Class and ID taken together define UBX-CFG-VALSET.

We can think of 0xB5 is the LSB although these bytes are not numbered. They are sent little-endian.

`0x09`	Length (LSB)
`0x00`	Length (MSB)

Note that the number of bytes in the payload, Length, is sent with the Least Significant Byte first and the Most Significant Byte second. We will later explain where this number comes from.

Next, we have our first look at a payload. Page 55 of the Interface Description provides some explanation, but I didn't find it entirely clear.

Byte offset		
0	0x00	version: it is 0x00
1	0x01	layer: RAM

In looking at the driver written by Paul Clark of Sparkfun, I now understand that a value of 0x01 means RAM, 0x02 means battery-backed-up RAM, and 0x08 is for EEPROM (aka flash).

2	0x00	transaction and action (transactionless in this case)
3	0x00	reserved

Next comes the configuration data. From page 141 I see

CFG-NAVSPG-DYNMODEL 0x20110021 E1 - - Dynamic platform model. This too is sent little-endian:

4	0x21	configuration data
5	0x00	
6	0x11	
7	0x20	

And finally, we have the value

8	0x06	which means AIR1

Oh no! We just fell into another area of confusion. As described in 4.5 Configuration data, this is where we define the location of the data we wish to address. In our example, we want to set the value.

Going back to the Table Of Contents, find 4.9 Configuration reference. These may look like commands, but they are payloads.

Scroll down to 4.9.12 CFG-NAVSPG: Standard precision navigation configuration

and you will find

CFG-NAVSPG-DYNMODEL 0x20110021 E1 - - Dynamic platform model

Why look here? Because I was tipped off that for the M10 to operate at 100,000 feet, I needed to be in *AIR1*. Without such direction, you are forced to sift through all Configuration items and related tables to find what you want.

Ah, another complaint. While each section of the Interface Description starts with the heading, the author chose to place the headings at the ***bottom*** of each table with an unclear delimiter between tables. If you go to page 143, you can see where it says

Table 22: Constants for CFG-NAVSPG-UTCSTANDARD

Below is the constant I need: AIR1. Never mind that there is paltry information on what AIR1 means in the M10's Interface Description. I then worked backward to see who referenced Table 22 and found that the Configuration item is

CFG-NAVSPG-UTCSTANDARD

Crash and burn. I was actually looking at Table 23, which is referenced by

CFG-NAVSPG-DYNMODEL

And you wonder why I have gray hair?

The last byte is the value:

0x06 **this is the value that corresponds to AIR1**

While looking through the u-blox 6 GPS/GLONASS/QZSS Receiver Description Including Protocol Specification V14

I stumbled into this table:

Dynamic Platform Model Details

Platform	*Max Altitude [m]*	*MAX Horizontal Velocity [m/s]*	*MAX Vertical Velocity [m/s]*	*Sanity check type*	*Max Position Deviation*
Portable	12000	310	50	Altitude and Velocity	Medium
Stationary	9000	10	6	Altitude and Velocity	Small
Pedestrian	9000	30	20	Altitude and Velocity	Small
Automotive	6000	84	15	Altitude and Velocity	Medium
At sea	500	25	5	Altitude and Velocity	Medium
Airborne <1g	50000	100	100	Altitude	Large
Airborne <2g	50000	250	100	Altitude	Large
Airborne <4g	50000	500	100	Altitude	Large

Amazing! It tells me what Airbore < 1g means. Well, it almost tells me the entire meaning: What does a Max Position Deviation of “Large” mean?
Putting what we have together:

```
uint8_t ubx_cfg_valset_dyn6[] = { // Series 9 and 10
0xB5,0x62,0x06,0x8A,0x09,0x00, // Header/Command/Size UBX-CFG-VALSET (RAM)
0x00,0x01,0x00,0x00,0x21,0x00,0x11,0x20,0x06, // Payload data (0x20110021 CFG-NAVSPG-DYNMODEL = 6)
```

Notes: Clive1 calls 0x06 0x8a the command but the Interface Description calls it class and ID. He also calls the Length the Size.

`0x00,0x01,0x00,0x00` is Version 0x00, RAM 0x01, reserve 0x00, reserve 0x00 as explained on page 7.

We can now verify that the Length is correct. The Length includes all bytes after the Length and before the checksum.

```
0x00,0x01,0x00,0x00,0x21,0x00,0x11,0x20,0x06
```

Which is 9 bytes. In hex, this is 0x00 0x09 but since it is sent little-endian, we see 0x09 0x00.

All that is left is the checksum.

The last two bytes are a Fletcher checksum that generates two bytes, CK_A and CK_B. See 3.4 UBX checksum for details.

The checksum is calculated over the Class, ID, Length, and Payload. It does not include the Preamble (aka header: 0xb5 0x62) or the two checksum bytes.

```
0x06,0x8A,0x09,0x00,0x00,0x01,0x00,0x00,0x21,0x00,0x11,0x20,0x06
```

Paul Clark's code covers every variation found in the Interface Description, so it makes sense for him to calculate the Fletcher Checksum in his code. Clive1 is dealing with a static array of numbers so he did the calculation off-line and hardcoded the checksum.

I understand from clive1 that the checksum is usually calculated at the time of transmission and is not stored.

I used Excel to generate the Fletcher Checksum values. Here are my equations:

	A	C	F	G	H
1	**HEX value**	**decimal value**	**decimal CKA**		**decimal CKB**
2	6	=HEX2DEC(A2)	=C2		=F2
3	8A	=HEX2DEC(A3)	=MOD(C3+F2,256)		=MOD(F3+H2,256)
4	9	=HEX2DEC(A4)	=MOD(C4+F3,256)		=MOD(F4+H3,256)
5	0	=HEX2DEC(A5)	=MOD(C5+F4,256)		=MOD(F5+H4,256)
6	0	=HEX2DEC(A6)	=MOD(C6+F5,256)		=MOD(F6+H5,256)
7	1	=HEX2DEC(A7)	=MOD(C7+F5,256)		=MOD(F7+H6,256)
8	0	=HEX2DEC(A8)	=MOD(C8+F7,256)		=MOD(F8+H7,256)
9	0	=HEX2DEC(A9)	=MOD(C9+F8,256)		=MOD(F9+H8,256)
10	21	=HEX2DEC(A10)	=MOD(C10+F9,256)		=MOD(F10+H9,256)
11	0	=HEX2DEC(A11)	=MOD(C11+F10,256)		=MOD(F11+H10,256)
12	11	=HEX2DEC(A12)	=MOD(C12+F11,256)		=MOD(F12+H11,256)
13	20	=HEX2DEC(A13)	=MOD(C13+F12,256)		=MOD(F13+H12,256)
14	6	=HEX2DEC(A14)	=MOD(C14+F13,256)		=MOD(F14+H13,256)
15		hex values	=DEC2HEX(F14)		=DEC2HEX(H14)

Note that I used MOD( ) to provide the same modulo 256 as using single bytes.

	A	C	E	F	G	H
1	**HEX v**	**decimal v**		**decimal CK**		**decimal CKB**
2	6	6		6		6
3	8A	138		144		150
4	9	9		153		47
5	0	0		153		200
6	0	0		153		97
7	1	1		154		251
8	0	0		154		149
9	0	0		154		47
10	21	33		187		234
11	0	0		187		165
12	11	17		204		113
13	20	32		236		93
14	6	6		242		79
15		hex values		F2		4F

←So CKA is 0xF2 and CKB is 0x4F.

The complete sequence of bytes sent to the M10 to change to AIR1 is:
{0xB5,0x62,0x06,0x8A,0x09,0x00,0x00,0x01,0x00,0x00,0x21,0x00,0x11,0x20,0x06,0xF2,0x4F};

Next, we need to prepare to understand the ACK that the M10 should return.

Refer back to page 49, 3.9.1 UBX-ACK-ACK (0x05 0x01).

I sent the sequence of bytes, and the M10 responded with

B5 header FOR ACK
62 header for ACK
05 class for ACK
01 ID for ACK-ACK (0 for ACK-NAK)
02 length of payload which is 2 bytes, little-endian
00 length
06 payload: Class ID of the Acknowledged VALSET Message
8A Message ID of the Acknowledged VALSET Message
98 CK_A correct values given above bytes without header
C1 CK B

Therefore, the ACK-ACK is as expected. It acknowledged my VALSET message and the checksum was correct, so no bits were corrupted.

As a final test, I will read back my new value using VALGET:

3.10.4 UBX-CFG-VALGET (0x06 0x8b)

From page 54 we have:

Header	Class ID	Length (Bytes)	Payload	Checksum
0xb5 0x62	0x06 0x8b	4 + [0..n]·4	see below	CK_A CK_B

The payload is not exactly the same as VALSET:

Byte	type	name	value	description
0	**U1**	**version**	**0x00**	
1	**U1**	**layer**	**0x00**	**for RAM (it was 0x01 in VALSET)**
2	**U2**	**position**	**0x00**	**skip this many key values**
3	**U2**	**position**	**0x00**	
4+4n	U4	keys	0x20	CFG-NAVSPG-DYNMODEL
			0x11	
			0x00	
			0x21	

Therefore, the payload is 8 bytes long.

Note: it shows the keys as **U4** which means they are unsigned little-endian 32-bit integer, so each key is sent with the Least Significant Byte first. I wonder if the U1 shown on page 55 for cfgData should be a U4.

The array that we will send to the M10 is

```
byte ubx_cfg_valget_dynmodel[] = { 0xB5,0x62,0x06,0x8b,0x08,0x00, // Header/Command/Size  UBX-CFG-VALGET (RAM)

0x00,0x00,0x00,0x00, 0x21,0x00,0x11,0x20, // key (0x20110021 CFG-NAVSPG-DYNMODEL)

0xeb,0x57}; //Fletcher checksum from my EXCEL spreadsheet
```

In 3.10.4.2 Configuration items we find the format of the response ***from*** the M10:

Byte	type	name	value	description
0	U1	version	0x01	note the change in value
1	U1	layer	0x00	retrieves data from the RAM layer

2	U2	position	0x00	number of configurations to skip
3		position	0x00	
4+n	U1	configuration data		key and value pairs.

I received back:
ACK response is B5 HEADER for UBX-CFG-VALGET message
ACK response is 62 HEADER
ACK response is 6 CLASS
ACK response is 8B ID
ACK response is 9 LENGTH
ACK response is 0 LENGTH
PAYLOAD
ACK response is 1 version 1 as per 3.10.4.2
ACK response is 0 RAM layer
ACK response is 0 positions to skip
ACK response is 0 positions to skip
ACK response is 21 KEY (it is E4 little-endian but docs says U1)
ACK response is 0
ACK response is 11
ACK response is 20
ACK response is 6 the pair value returned which means AIR1
ACK response is F3 CKA
ACK response is 5C CKB
ACK response is B5 HEADER this is a UBX-ACK-ACK message
ACK response is 62 HEADER
ACK response is 5 CLASS
ACK response is 1 ID
ACK response is 2 LENGTH LSB
ACK response is 0 LENGTH MSB
ACK response is 6 class of GETVAL message being acknowledged
ACK response is 8B ID of GETVAL message being acknowledged
ACK response is 99 CKA
ACK response is C2 CKB

Sending M10 a GETVAL command returned the data plus an ACK for the GETVAL command. I wasn't expecting to get an ACK since I did get the data.

u-blox neo M6

I tried to apply what I had learned from the M10 to sending a message to my u-blox neo M6 using the M6 Interface Description but did not get an ACK. At 3 for $15, I've always suspected these devices were clones with part of the functionality never implemented. Without a genuine M6, I can't know if I made a coding error or if the hardware is at fault.

Acknowledgments

Thanks to PaulZC (Paul Clark) for his well-written code. Thanks to clive1 for his concrete example that unlocked all of the mystery plus comments on this article. Both experts were found on the Sparkfun Forum.

I welcome your comments and questions.

If you want me to contact you each time I publish an article, email me with "Subscribe" in the subject line. In the body of the email, please tell me if you are interested in metalworking, software plus electronics, kayaking, and/or the Lectric XP eBike so I can put you on the right distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line. No hard feelings.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org