Document prise en main d’un lIDAR A1M8

I) Hardware part

Connect LiDAR like that :

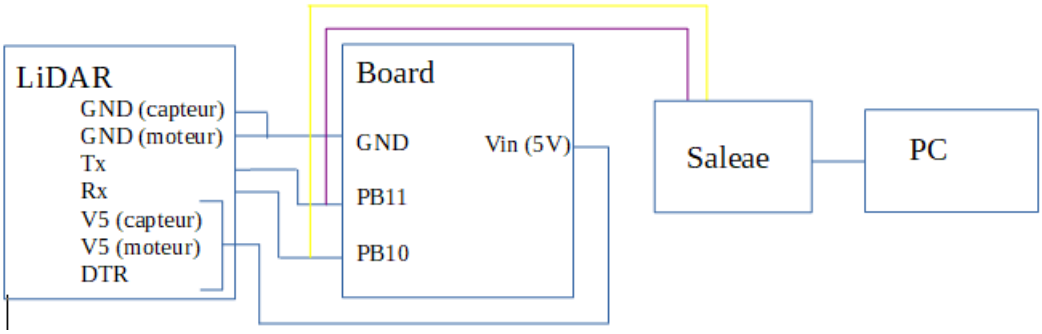


Figure 1 : Assembly schematics.

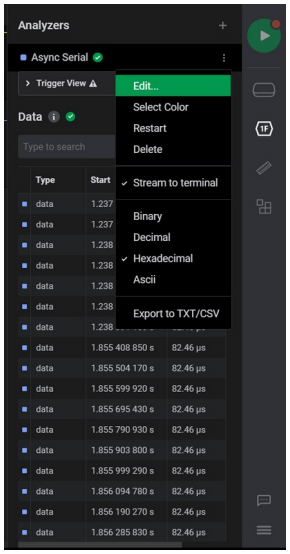
For Saleae connection, you can use this :

https://fr.banggood.com/40Pcs-Test-Clamp-Wire-Hook-Test-Clip-for-Logic-Analyzer-p-963539.html?utm_source=googleshopping&utm_medium=cpc_organic&gmcCountry=FR&utm_content=minha&utm_campaign=minha-frg-fr-pc&currency=EUR&cur_warehouse=CZ&createTmp=1

Now, download, logic2 app, available here :

https://logic2api.saleae.com/download?os=windows

In the section « Async Serial », select « Edit » :



And set the baudrate to 115200.

So, you’re ready to observe frame between your board and sensor.

II) Software part

Here we will discuss about how to send the 8 request available on this LiDAR and their interpretations.

Tableau 1 : All commands available in A1M8 model

Single request without response	Single request with single response	Single request with multiple response
STOP : Stops a point acquisition.	GET_LIDAR_INFO : Provides information about the liDAR such as the model (RPLiDAR A1M8), firmware (1.20), hardware and serial number.	FORCE_SCAN : Forces the LiDAR to do a scan, if the LiDAR bug because it gets stuck on an erroneous value, this allows to debug it.
RESET : Restart the LiDAR and empty the buffer.	GET_DEVICE_HEALTH : Lets you see if LiDAR has a problem and what the problem is.	SCAN : This is the “standard” scan with a scan frequency of 2000 points/s.
	GET_SAMPLERATE :Provides the duration of a single acquisition.	EXPRESS_SCAN : This is the “fast” scan with a scan frequency of 4000 points/s.
		BOOST_SCAN : This is the “super fast” scan with a frequency of 8000 points/s (but the firmware of the A1 model is too old).

For my part, I chose to process these 8 orders in 4 different categories:

- Request without response (STOP, RESET)
- Request with single response.
- Request with multiple responses and no payload required.
- Request with multiple responses and requiring a payload (EXPRESS_SCAN)

Indeed, the last category requires a special sending protocol. Finally, each LiDAR response is divided into 2 parts:

- The "Response Descriptor" which contains informations relating to the "Byte offset" such as the size, the sending mode….
- The "Byte offset" which contains the information that the user wishes to receive with his request.

1) Request with single response

Before starting, I precise that each request packet is initialized at the beginning of the main().

```
 98   /* USER CODE END 2 */
 99   /* Infinite loop */
100   /* USER CODE BEGIN WHILE */
101   rplidar_cmd_packet_t env1; //STOP send packet
102   rplidar_cmd_packet_t env2; //RESET send packet
103   rplidar_cmd_packet_t env3; //GET_DEVICE_INFO send packet
104   rplidar_cmd_packet_t env4; //GET_DEVICE_HEALTH send packet
105   rplidar_cmd_packet_t env5; //SAMPLERATE send packet
106   rplidar_cmd_packet_t env6; //FORCE_SCAN send packet
107   rplidar_cmd_packet_t env7; //SCAN send packet (2000 pt/s => 360 pt/tour)
108   rplidar_cmd_packet_t env8; //EXPRESS_SCAN send packet (4000 pt/s => 720 pt/tour)
```

Figure 2: Declaration of all request packet.

A) Request GET_DEVICE_INFO

1) Response descriptor

```
130   /*commands with single response*/
131
132   //GET_DEVICE_INFO command in order to receive all information about LiDAR (model, firmware, serial number, Hardware version)
133   env3.syncByte=RPLIDAR_CMD_SYNC_BYTE;
134   env3.cmd_flag= RPLIDAR_CMD_GET_DEVICE_INFO;
135   env3.size=0x00;
136   uart_send(&huart3,(uint8_t *) &env3, sizeof(env3));
```

Figure 3 : Initialization of send packet.

Uncomment fig.3 code and try to understand, by looking in the Interface Protocol and Application Notes document, how to build a request packet. It should be noted that all commands and responses we will address follow this “structure”:

Figure 4 : Request Packet.

Figure 5: Response Descriptor packet.

The request packet structure, taken from the mentioned documentation, is defined like a structure in the *lidar_protoc.h*:

```
47  typedef struct _rplidar_cmd_packet_t {
48      uint8_t syncByte; //must be RPLIDAR_CMD_SYNC_BYTE
49      uint8_t cmd_flag;
50      uint8_t size;
51      uint8_t data[0];
52  } __attribute__((packed)) rplidar_cmd_packet_t;
```

Figure 6 : Definition of structure in header « lidar_protoc.h ».

All fields are explained and the hexa codes of all requests are found in the lidar_cmd file.h:

```
// Commands
//-----------------------------------------

// Commands without payload and response
#define RPLIDAR_CMD_STOP                0x25
#define RPLIDAR_CMD_SCAN                0x20
#define RPLIDAR_CMD_FORCE_SCAN          0x21
#define RPLIDAR_CMD_RESET               0x40


// Commands without payload but have response
#define RPLIDAR_CMD_GET_DEVICE_INFO     0x50
#define RPLIDAR_CMD_GET_DEVICE_HEALTH   0x52

#define RPLIDAR_CMD_GET_SAMPLERATE      0x59 //added in fw 1.17

// Commands with payload and have response
#define RPLIDAR_CMD_EXPRESS_SCAN        0x82 //added in fw 1.17
```

Figure 7 : « lidar_cmd.h »

We can see that the request associated to the GET_DEVICE_INFO is linked to the hexadecimal code 0x50, as mentioned here:

Figure 8 : GET_INFO structure.

It is also much easier to understand how the structure of fig.4 is defined.
Warning: All commands always start with the 0xA5 starter flag!
Once the principle is understood, let's send request to LiDAR and observe its response on logic 2

Figure 9 : Response of LiDAR on logic2.

We see that the LiDAR response is divided in 2 sections, the first that we will deal in this part and the second that we will see after.

So, comparing to fig.5, the response descriptor (see in fig.9):

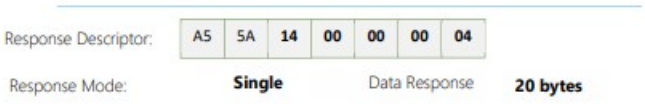
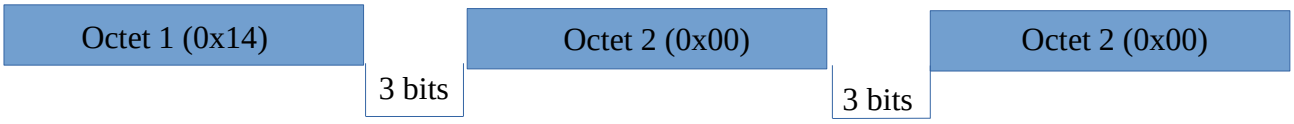


Figure 10 : Response Descriptor of GET_INFO.

We find at the beginning our 2 bytes, which are our request starter flag (0xA5), followed by the response starter flag (0x5A) as mentioned in fig.5.

Then, we find our 3 bytes corresponding to the "Data Response Length". In fact, in fig.5, it talk about 30 bits reserved. However, it includes the 3 bits transitions between our 3 byes.



Then, we receive the "Send Mode" on 1 byte, here 0x00. This corresponds to our two sending modes:

- Single request /Single Response (defined as “Single” on Fig.10)
- Single request /Multiple response (defined as “Multiple”)

However, in fig.11, only two bits are necessary to identify the mode, as defined below:

Send Mode	Description
0x0	Single Request – Single Response mode, RPLIDAR will send only one data response packet in the current session.
0x1	Single Request – Multiple Response mode, RPLIDAR will continuously send out data response packets with the same format in the current session.

Figure 11 : Tous les modes d’envoie.

Here we have 0x00 which is the first mode.

Warning: If a request correspond to the second mode, we will observe 0x40 in the frames instead of 0x01. Indeed, 0x40 is $(0100\ 0000)_2$. But, send mode is on 2 bits on fig.5, it is actually the two MSB that you have to take.
Thus, we find 0x01….

Finally, there is the "data type", which describes how to interpret data returned by the LiDAR. Indeed, LiDAR responds to 6 commands and system code in binary various informations that don’t have the same meanings. Also, the information must be able to be retrieved through this byte, which indicates how to interpret data received.

2) Byte Offset

Let’s now turn to the second section of the LiDAR response called byte offset.

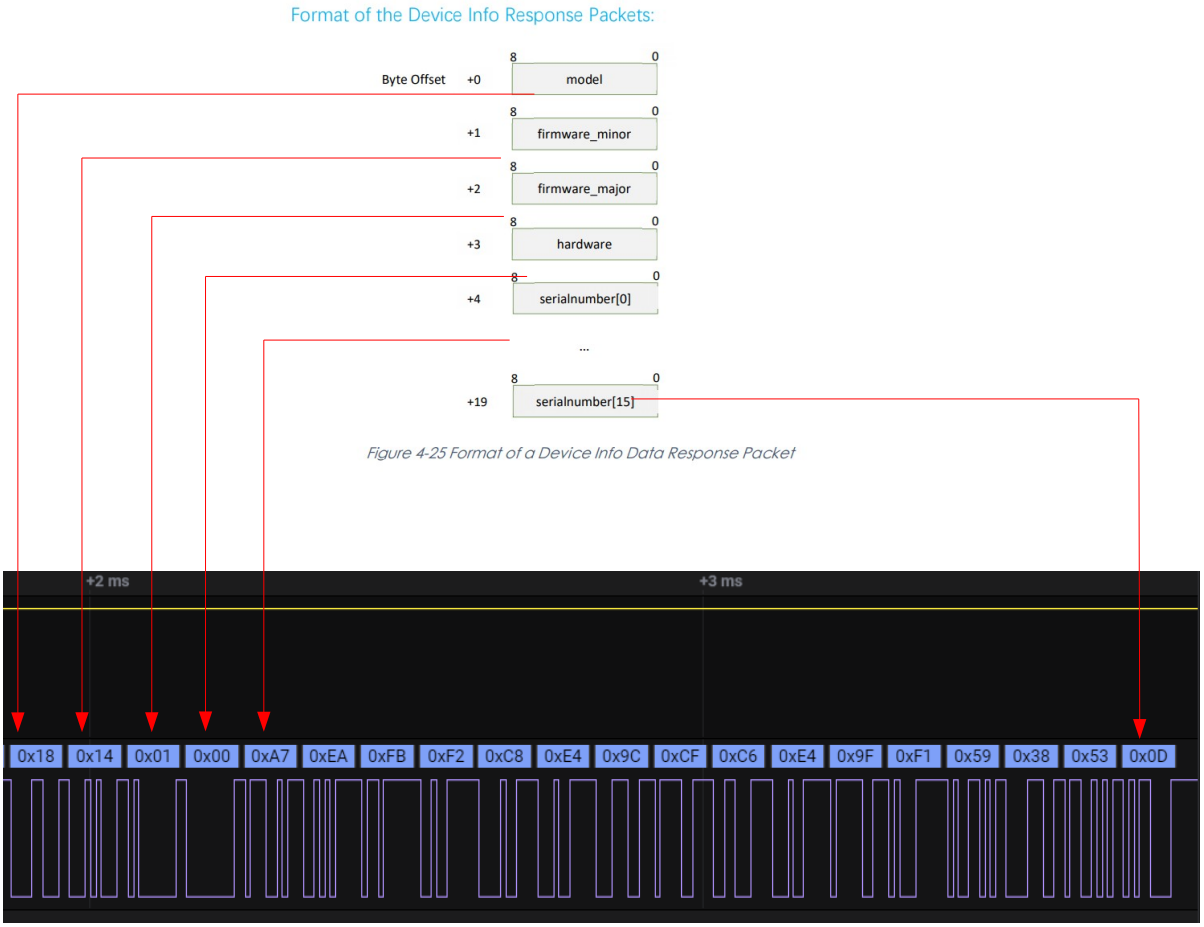


Figure 4-25 Format of a Device Info Data Response Packet

Figure 12 : Byte offset structure with corresponding byte on logic2.

Furthermore, we see that we have 20 bytes in the "byte_Offset", but $20=(00010100)_2=(0x14)_{16}$. It corresponding to "Data Response Length" announced in Fig.10.

B) Request « GET_DEVICE_HEALTH »

Uncomment the corresponding code (and comment the previous section):

```
138  //GET_DEVICE_HEALTH command in order to obtain complete diagnostic of LiDAR
139  env4.syncByte=RPLIDAR_CMD_SYNC_BYTE;
140  env4.cmd_flag=RPLIDAR_CMD_GET_DEVICE_HEALTH;
141  env4.size=0x00;
142  uart_send(&huart3,(uint8_t *) &env4, sizeof(env4));
```

Figure 13 : Send packet definition.

To pass quickly on it, we have exactly the same definition of structure than part A. Our request is set for 0x52.

1) Response descriptor

Figure 14 : Response descriptor associated.

Let’s compare to the signal observed on logic2:

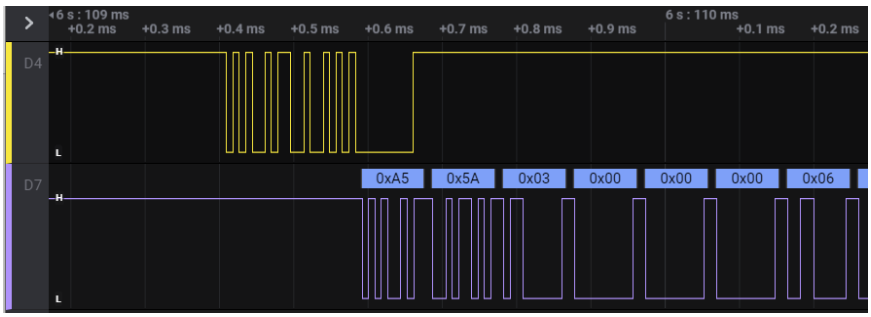


Figure 15 : frames on logic2.

2) Byte Offset

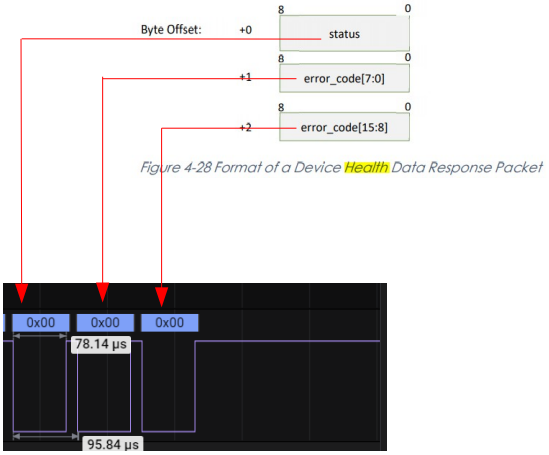


Figure 16 : Structure of byte offset with corresponding byte.

Let’s see the signification of different status :

Field Name	Description	Examples / Notes
status	RPLIDAR Health State	Value definition: 0: Good 1: Warning 2: Error When the core system detects some potential risk that may cause hardware failure in the future, the status value will be set to Warning(1). But RPLIDAR can still work as normal. When RPLIDAR is in the Protection Stop state, the status value is set to Error(2).

Figure 17 : Status meaning.

Also, our LiDAR returns 0 for status and error codes, so everything is OK.
In the case of a "Warning" or "Error" status, the LiDAR will specify in the 2 next bytes the reason why it isn't good. I would like to give you an overview of the meaning of each error code but the documentation doesn't mention the meaning of the error codes and the technical support did not want to communicate it to me.

C) Request « GET_SAMPLERATE »

Uncomment the following code et comment previous lines :

```
144  //GET_SAMPLERATE command in order to obtain the duration of single acquisition.
145  env5.syncByte=RPLIDAR_CMD_SYNC_BYTE;
146  env5.cmd_flag=RPLIDAR_CMD_GET_SAMPLERATE;
147  env5.size=0x00;
148  uart_send(&huart3,(uint8_t *) &env5, sizeof(env5));
```

Figure 18 : Definition of request packet.

1) Response descriptor

Response Descriptor:	A5	5A	4	00	00	00	15

Response Mode: **Single** Data Response Length: **4 bytes**

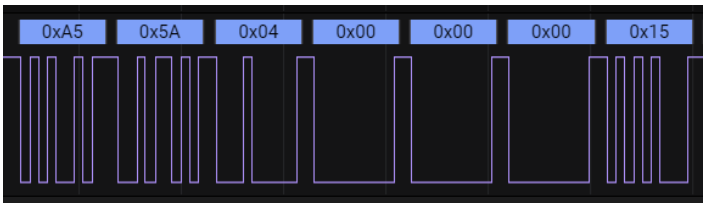


Figure 19 : Response descriptor with frames observed on logic2.

2) Byte Offset

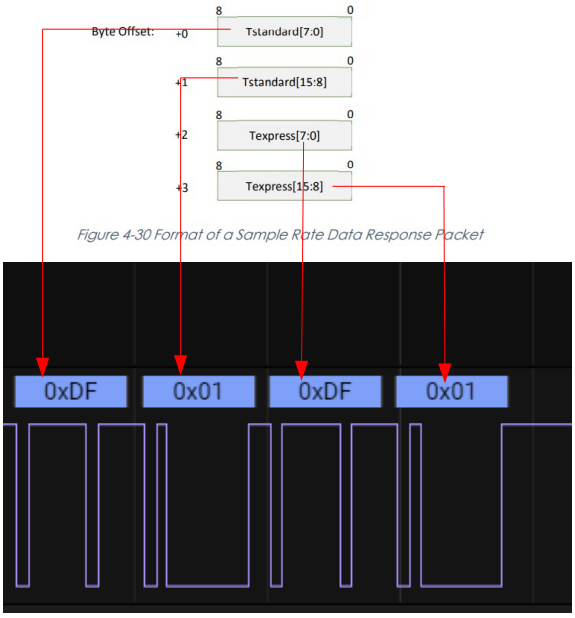


Figure 20 : Byte offset on logic2.

To understand this, refer to tab.1 and the fact that only two scan modes are available: standard (2000 pt/s) and express (4000 pt/s). Also, first two bytes mean the acquisition time of one point in standard mode and the next two for express mode.

Therefore, Tstandard[15:0] = 0x01DF = $(0000\ 0001\ 1101\ 1111)_2$=479µs

Thus, LiDAR (in standard scan) acquires one point on 479µs. So, a simple calculation allows us to see that in reality the LiDAR acquires about 2088pt/s unlike the 2000pt/s announced.

Similarly, the scan time for the express scan is identical, except that it is in Q1 fixed point format:

Therefore, Texpress[15:0] = 0x01DF = $(0000\ 0001\ 1101\ 1111)_2$=479µs

But in Q1, we take into account a dot to the left of the LSB (equivalent to divided by two the value in decimal). So, the LiDAR takes 239.5µs to acquire 1 point, we redo a calculation that allows us to see that in express mode the LiDAR actually acquires 4175pt/s.

2) Request with multiple responses and without payload.

Let’s look at the request to do a scan. FORCE_SCAN and SCAN are identical in execution, I voluntarily choose to skip the first request to speak directly about the standard scan.

A) Request « SCAN »

Uncomment the following lines:

```
158     //Envoie d'une commande scan pour avoir la map en balayage "standard", attention si le lidar est en "protection stop state" faire u
159     uint8_t buf_void;
160     env7.syncByte=RPLIDAR_CMD_SYNC_BYTE;
161     env7.cmd_flag=RPLIDAR_CMD_SCAN;
162     env7.size=0x00;
163
164     uart_send(&huart3,(uint8_t *) &env2, sizeof(env2));//Envoi d'un reset (décommenter la commande associée), cela permet de redémarrer
165
166     uart_receive(&huart3,(uint8_t *) &buf_void, sizeof(buf_void));//On attends la réception d'un octet qui confirme bien le redémarrage
167
168     uart_send(&huart3,(uint8_t *) &env7, sizeof(env7));//On envoie la commande du balayage standard, attention si un balayage est déjà
169                                                        //la nouvelle demande de balayage ne prendra pas effet, recompiler une seconde
```

Figure 21 : Code for sending scan request.

First, I send a RESET, it’s a sort of restart. Without it, sensor doesn’t react to the scan request. Indeed, if you are interested in the LiDAR’s working states:

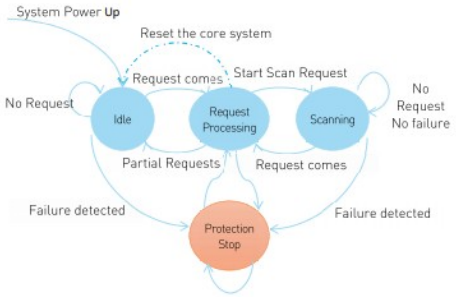


Figure 3-1 RPLIDAR’ s Major Status Translation

The Idle state is the default state of RPLIDAR which will be entered automatically after powering up or reset. Both the laser diode and the measurement system are disabled in this state, and the whole system is in power saving mode. Once RPLIDAR enters the Scanning state, the laser diode and the measurement system will be enabled and RPLIDAR will start measuring distance and sending the result out continuously.

Figure 22 : LiDAR’s working states.

We can see that the "Idle" mode is set by default, so no scan request is taken into account unless a reset is sent. However, if the scan is sent immediately after the reset request, then the hardware doesn’t have time to restart, so in order to delay, I wait to receive 1 byte to be sure that the HW has restarted before sending scan request. Indeed, when the LiDAR receives a RESET, it returns something that’s not really response. Indeed, the LiDAR will return:

« RP LiDAR SYSTEM.

FIRMWARE VER 1.20 – rtm, HW VER 0

MODE : 18 »

This is similar to the restart screen of a computer that displays WINDOWS 10…. But I use it to make sure the LiDAR is reset.

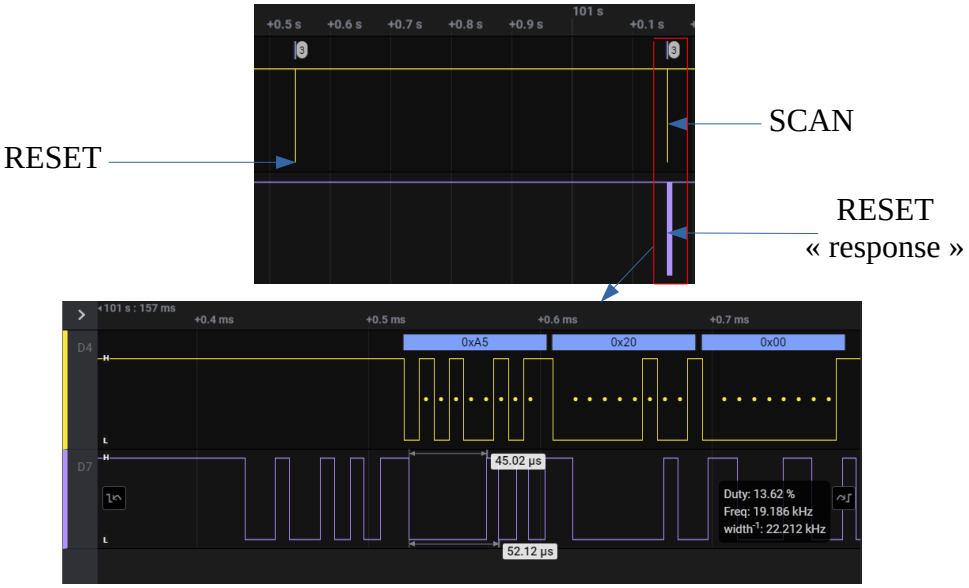


Figure 23 : Send SCAN request after 1 byte.

1) Response Descriptor

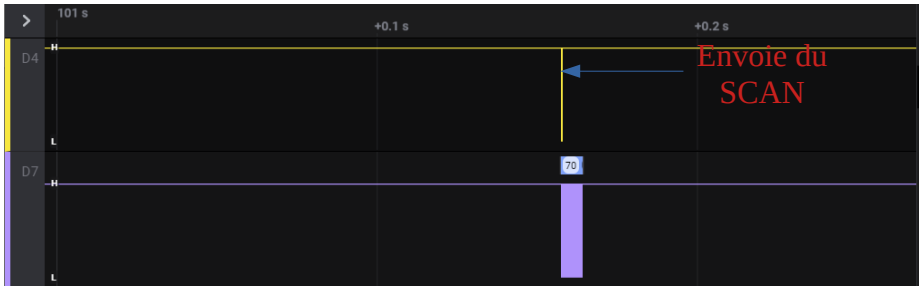
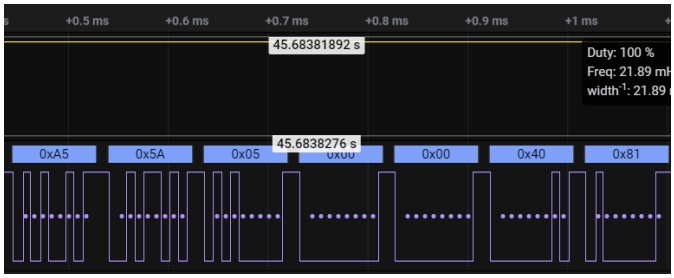


Figure 24 : Quantity of Bytes receive.

I specify that the message returned by the LiDAR for a RESET, is on 63 bytes. So our response descriptor is from the 64th bytes. Zooming in:



Response Descriptor:	A5	5A	05	00	00	40	81

Response Mode: **Multiple** Data Response Length: **5 bytes**

Figure 25 : Response Descriptor with logic2 frames.

2) Byte Offset

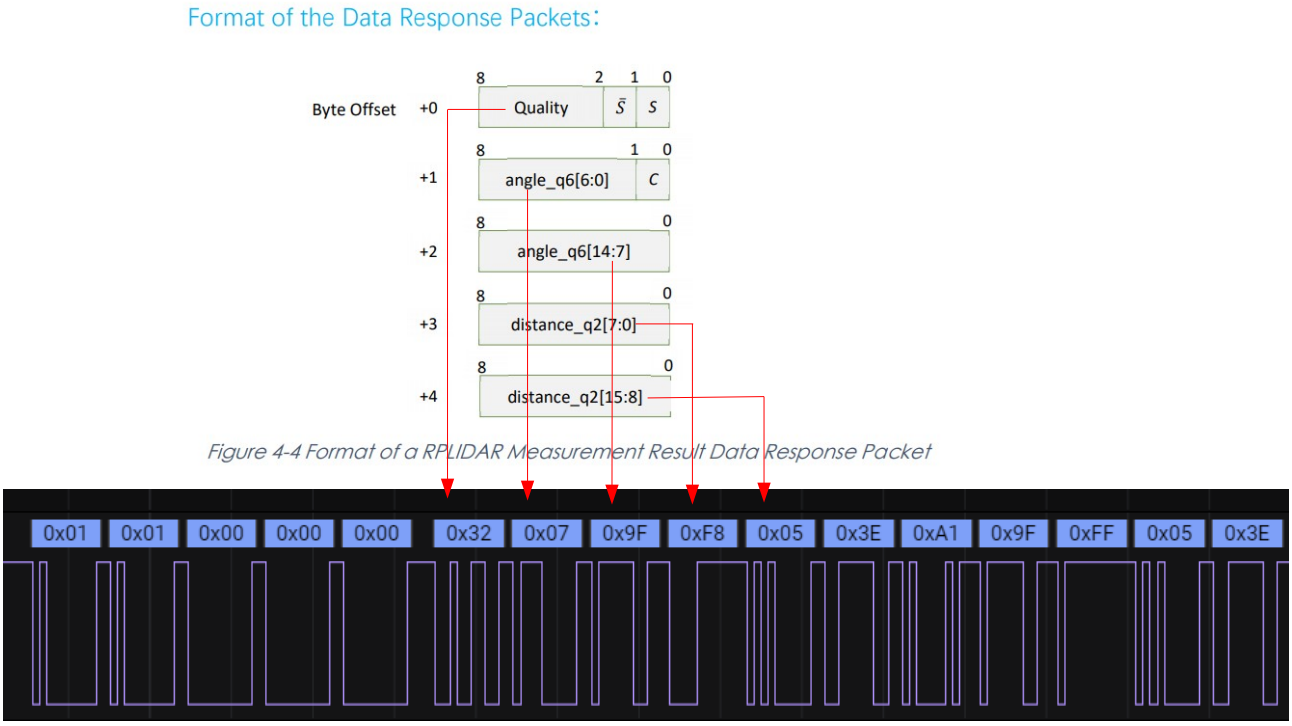


Figure 26 : SCAN's byte offset.

Note that the offset byte of the SCAN (express or standard) will always start with an erroneous point, signified by 5 bytes «0x01» «0x01» «0x00» «0x00» «0x00», in standard mode. So let's look at the first point detected by the LiDAR and therefore at the first 5 bytes that describe it:

Quality : 0x32 = $(0011\ 0010)_2$

The quality of a measurement is related to the intensity of the ray received by the LiDAR. Indeed, when a beam is emitted, the ideal scheme would be for it to fully reflect on an obstacle and return to LiDAR. In reality, part of the ray comes back well but the other part will bounce back on other obstacles and comes back to the LiDAR. Sensor having continued to rotate, he will think that this residual ray corresponds to an obstacle detected at its current angle, the result is an erroneous value on the map. So, the LiDAR assigns a rating ranging from 0 to 63 (00111111 the ideal case of total radius reflection) to allow the user, if he wishes, to filter these residual values for a better map.

So, look the quality byte :

q[5]	q[4]	q[3]	q[2]	q[1]	q[0]	!S	S
0	0	1	1	0	0	1	0

Thus, q[7:0] = $(0000\ 1100)_2$ = 12/63

Be careful, we must not infer that the quality is low because it depends on the environment. Indeed, if the environment has several obstacles which disrupt IR rays of LiDAR, then it can be considered a good quality.

Angle : 0x9F07 = $(1001\ 1111\ 0000\ 0111)_2$

angle_q6[14]	angle_q6[13]	angle_q6[12]	angle_q6[11]	angle_q6[10]	angle_q6[9]	angle_q6[8]	angle_q6[7]	angle_q6[6]	angle_q6[5]	angle_q6[4]	angle_q6[3]	angle_q6[2]	angle_q6[1]	angle_q6[0]	C
1	0	0	1	1	1	1	1	0	0	0	0	0	1	1	1

Which represent (add 0 for MSB) : angle_q6[15:0] = 20355

However, it is a Q6 fixed point format, so we must divide the decimal value by 64 (2^6 => shift the dot to 6 bits on the left). So we have an obstacle at 318.0469°.

Distance : 0x05F8 = $(0000\ 0101\ 1111\ 1000)_2$

distance_q2[15]	distance_q2[14]	distance_q2[13]	distance_q2[12]	distance_q2[11]	distance_q2[10]	distance_q2[9]	distance_q2[8]	distance_q2[7]	distance_q2[6]	distance_q2[5]	distance_q2[4]	distance_q2[3]	distance_q2[2]	distance_q2[1]	distance_q2[0]
0	0	0	0	0	1	0	1	1	1	1	1	1	0	0	0

Which represent : distance_q2[15:0] = 1528

However, it is in Q4 fixed point format, so we must divide the decimal value by 4 (2^2 => shift the dot to 2 bits on the left). So we have an obstacle at 382mm.

By looking at the orientation of the angles on the documentation, we visualize where the obstacle is located:

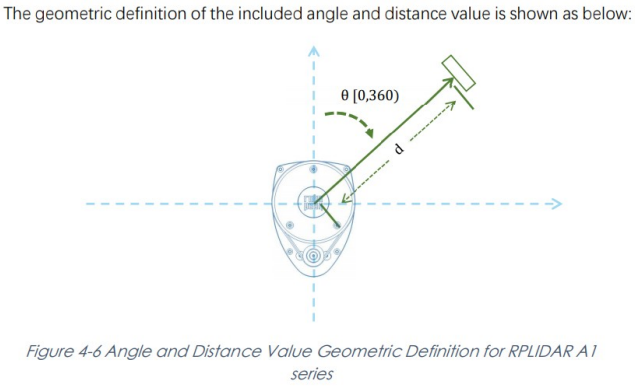


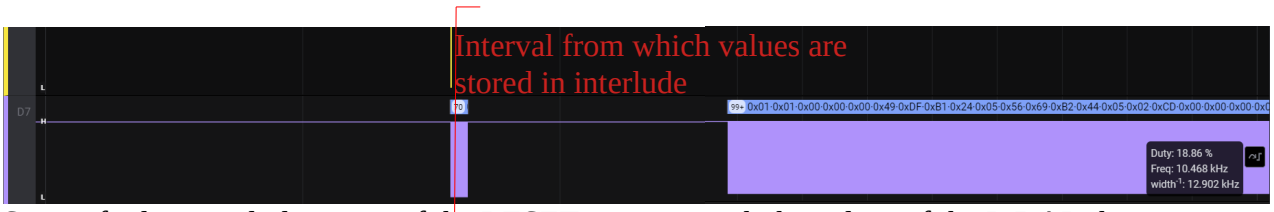
Figure 27 : Polar plot orientation.

Now, let's see how to get the values…. Uncomment the associated code section, in addition to the previous code:

```
174   /*Do interrupts until interlude table will fill before calling RxCpltCallback function*/
175   for(uint64_t i=0; i<sizeof(interlude); i++)
176   {
177     uart_receive_IT(&huart3,(uint8_t *) &interlude, sizeof(interlude));
178   }
179   while(1)
180   {
181     /* USER CODE END WHILE */
182     if (flag_LiDAR==1)//When flag is equal to 1, it's significate that I filled my interlude table with all value return by my LiDAR
183     {
184       uart_send(&huart3,(uint8_t *) &env1, sizeof(env1));//We send a STOP command in order to stop scan
185       flag_LiDAR=0;
186       break;
187     }
188     /* USER CODE BEGIN 3 */
189   }
```

Figure 28 : Code to receive distances and angles values.

Once the SCAN is sent, we enter in for loop which will generate an interruption each time bytes are received. The aim is to fill in an interlude table, defined in global. With this step, interlude is filled with all the values received after the SCAN, if we resume, we have:



So, we find in interlude, a part of the RESET message with the values of the LiDAR that interests us.

To resume the program, once interlude filled in, the code will execute my interrupt routine. During it, I activate a flag that I declared globally, in order to synchronize the main() with the execution of the last IT.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  flag_LiDAR=1;//use a flag to synchhronize the end of interrupt with main function
}
```

Once the flag was activated, I put in my while(1) the commands to execute in this case. In fact, it is simply a STOP command that will finish the SCAN, once the values are recovered. In terms of the number of values to isolate, 360° rotation of LiDAR corresponds to 360 points acquisitions. Now, a point is characterized by 5 bytes, so it is necessary that our table can store 360*5=1800bytes. Moreover, there are also a part of RESET « response » in interlude values. So, after tests, I can say that it stored exactly 72 waste values before the first byte that interests us. Therefore, this is why my interlude table has a size of 1872. This gives on logic2:

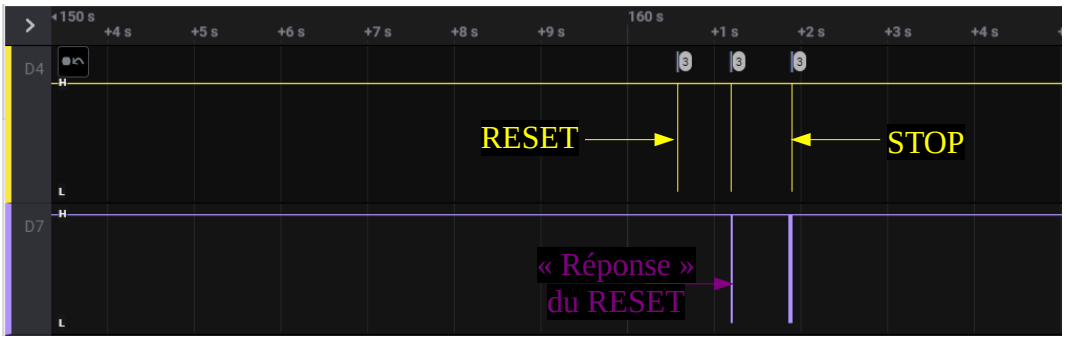


Figure 29 : Vue d'ensemble des signaux sur logic2.

Now, we need to isolate the LiDAR values in the byte_offset table declared like global variable. I specify that I could work in terms of number of bytes to go through interlude and start to store values in byte_offset from the 72th value, but this isn't good solution because if a send message is late or response, this solution will not work.

Otherwise, I think it is better to exploit the fact that the 5 bytes starting byte_offset, are "0x01" "0x01" "0x00" "0x00" "0x00", so I propose as a program:

uint32_t pos ;//Declare pos like global variable

uint8_t k = 0 ;//Declare k like local variable in the main()

```
uint8_t premier_octet = 2 ; //Declare premier_octet like local variable in the main()
uint8_t deuxieme_octet = 2 ; //Same thing
/*On écrit la suite, après la sortie du while(1)*/
for(uint32_t i=0 ; i<sizeof(interlude) ;  i++)//We are going through interlude[]
{
     if ((interlude [i] == 1)&&(premier_octet == 2))//if interlude = 1 which is the first value starting
                                                    //the byte offset
    {
       premier_octet = interlude[i] ; //We save the value
    }
    if ((interlude [i] == 1)&&(interlude[i-1] ==1))  //We ensure that the value received by the
                                                    //second_byte is the second 1 of the byte_offset
    {
       deuxieme_octet = interlude[i] ;
    }
    if ((premier_octet != 2) &&(deuxieme_octet != 2))
    {
       pos=i+3;//We advance the current position by 3 because they correspond to the next 3 bytes
                //at 0 of byte_offset
       break ;
     }
}
uint32_t j=0 ;
for(uint32_t i=pos ; i<sizeof(interlude) ;  i++)
{
         byte_offset[j]=interlude[i] ;
         j++ ;
}
```

It’s a suggested algorithm and I didn’t have time to test it, it may not work. Once our values of qualities, angles and distances are stored, it is necessary to fill our tables angles and distances that I declared in global. For this, I suggest this piece of code:

```
uint32_t k=0 ;// déclarer en locale dans le main
```

```
for(uint32_t i=5 ; i<sizeof(byte_offset) ;  i=i+5) //We are going through byte_offset by 5 because it
                                                    //is by pack of 5 that we have our characteristics
                                                   //for points
{
        angle[k] =( (byte_offset[i-4] + (byte_offset[i-3]<<8))>>1)<<6;
        distance[k] =  ((byte_offset[i-2] + (byte_offset[i-1]<<8)) )<<2;
        k++ ;
}
```

Once our angle and distance values are recovered, we can create the Lidar service using the correct Object Dictionnary to have the values in the desired unit and the Luos commands for our 8 request. So that's the point missing to my project.

3) Request with multiple responses and payload.

A) EXPRESS_SCAN

I precise that this part is an extra for my project, if you complete my code for the standard scan, it should be great for me and we could propose it to Luos company.

Uncomment the following code :

```
194  /*Command with multiple response and recquire a payload*/
195
196  //Send EXPRESS_SCAN command in order to obtain more acquisition (4000 pt/s)
197
198  //Definition of first packet for command
199  uint8_t buf_void_exp;
200  env8.syncByte=RPLIDAR_CMD_SYNC_BYTE;
201  env8.cmd_flag=RPLIDAR_CMD_EXPRESS_SCAN;
202  env8.size=0x05;
203
204  //Definition of second packet with payload
205  rplidar_payload_express_scan_t payload;//Définition of payload
206  payload.working_mode=RPLIDAR_EXPRESS_SCAN_MODE_NORMAL;
207  payload.working_flags=0x0000;
208  payload.param=0x0000;
209
210  //Checksum definition
211  uint8_t checksum; //Checksum definition
212  checksum=0x22;
213
214  uart_send(&huart3,(uint8_t *) &env2, sizeof(env2));//Need to reset LiDAR in
215
216  uart_receive(&huart3,(uint8_t *) &buf_void_exp, sizeof(buf_void_exp));//Rece
217
218  //Send first packet
219  uart_send(&huart3,(uint8_t *) &env8, sizeof(env8));//Envoi de la commande
220
221  //Send the payload
222  uart_send(&huart3,(uint8_t *) &payload, sizeof(payload));
223
224  //Send the checksum
225  uart_send(&huart3,(uint8_t *) &checksum, sizeof(checksum));
```

Figure 30 : Sending request+payload+checksum.

Note that there are two versions of the express mode in the documentation:

- Legacy mode to acquire 4000pt/s at a distance of 12 meters
- Extended mode which allows 4000pt/s to be acquired at a distance of 16 meters.

With the current firmware version, we only have access to the Legacy mode.

1) Response Descriptor

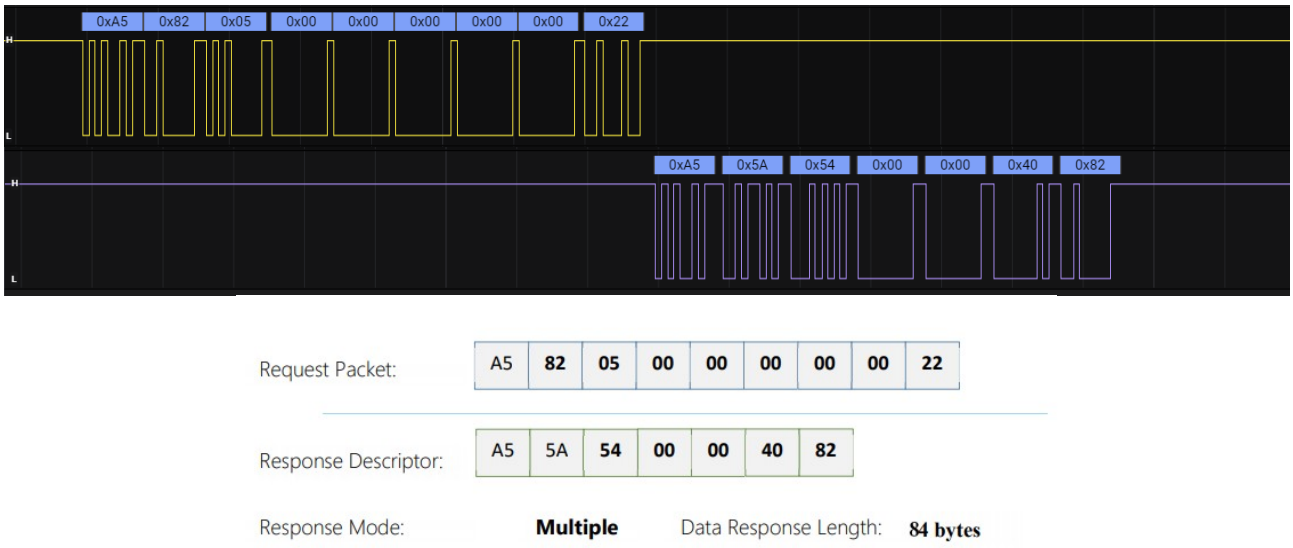


Figure 31 : Response Descriptor with corresponding frames.

I should point out that checksum is defined according to the documentation, as follows:

The checksum value can be calculated using the following equation:

$$checksum = 0 \oplus 0xA5 \oplus CmdType \oplus PayloadSize \oplus Payload[0] \oplus \dots \oplus Payload[n]$$

This is a byte to ensure that all bytes have been send by LiDAR correctly.

2) Byte_Offset

It should be noted that in standard scan, LiDAR captures the distance value and the angles very regularly (360 times/rotation) and shapes all of these to return only the results. In express mode, the LiDAR will save time to be able to follow the rythm of 4000pt/s. To do this, it will space the angle measurements and return the necessary variables to the user, in order to let him calculate the angle. In short, the sensor no spend time with the most expensive calculations in terms of time, it the user task. The angles must therefore be calculated according to this formula:

$$\theta_k = \omega_i + \frac{AngleDiff(\omega_i, \omega_{i+1})}{32} \cdot k - d\theta_k$$

Let’s detail it…. First of all, we must understand that in express, the data are organized in response packet as follows:

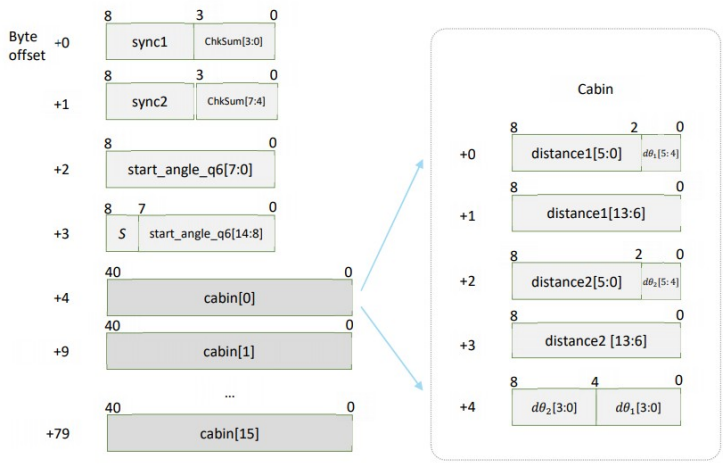


Figure 4-11 Format of a RPLIDAR Express Scan Data Response Packet (Legacy Version)

Figure 32 : Byte offset of LEGACY EXPRESS mode. A response packet is about 84 bytes, as we can see on logic2:

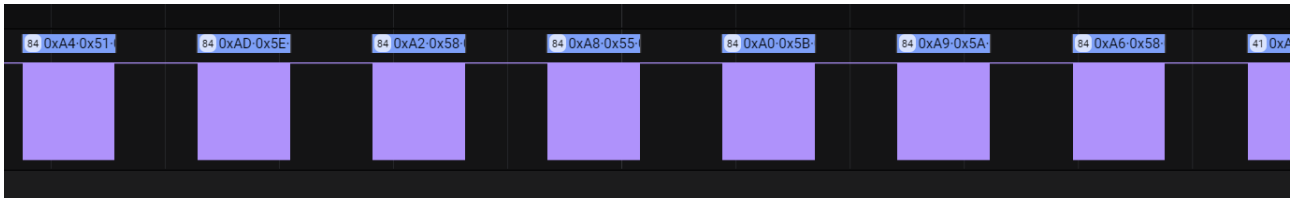


Figure 33: Responses packets.

If we zoom in on the first one:

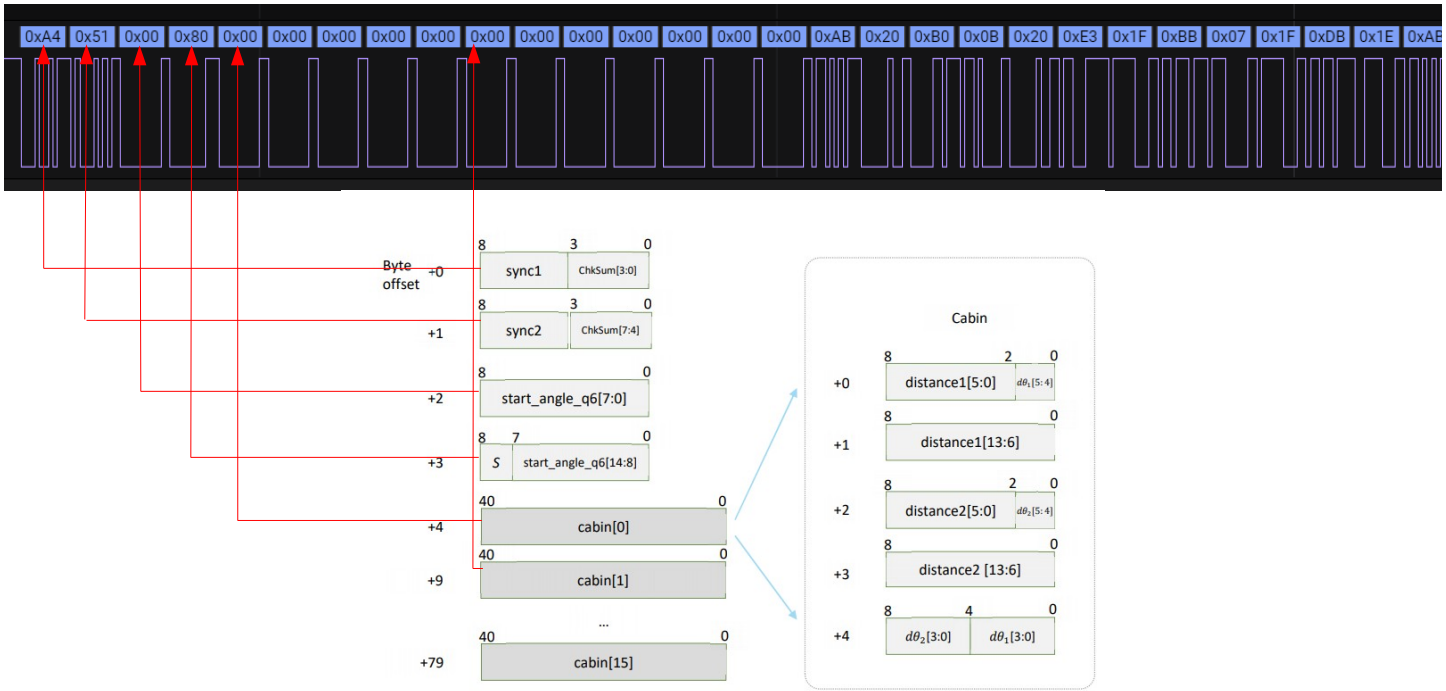


Figure 4-11 Format of a RPLIDAR Express Scan Data Response Packet (Legacy Version)

Figure 34 : Byte corresponding.

Sync1+ChkSum[3:0] :

Sync 1 refers to the Starter Flag that starts each LiDAR response, here A5 but if we only take the 4 MSB the result is «A» which is what we have. The 4 LSB are therefore for the first part of the checksum (here 4)

Sync2+ChkSum[7:4] :

Similarly, sync 2 is for the 4 MSB of the second starter flag, so 5 (taken from 5A). The 4 MSB of the checksum represent 1, giving us a checksum value equal to 0x14.

Start_angle_q6 :

Here, the start_angle means the starting angle of the packet, we talked earlier about time saving and this is what happen here, the LiDAR spaces its angle measure. In clear, we have one start_angle for for N points per packets. And for all of these N points, he will use the formula mentioned earlier to associate the right angles to the distances. This start_angle corresponds, in the formula, to ω_i.

So, we have : 0x8000 (S compris) :

S	angle_ q6[14]	angle_ q6[13]	angle_ q6[12]	angle_ q6[11]	angle_ q6[10]	angle_ q6[9]	angle_ q6[8]	angle_ q6[7]	angle_ q6[6]	angle_ q6[5]	angle_ q6[4]	angle_ q6[3]	angle_ q6[2]	angle_ q6[1]	angle_ q6[0]
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Giving us a 0° angle for the first packet. I should point out that when S is set to 1, this means that the LiDAR starts an acquisition for a new rotation. In other words, while S=0, the LiDAR will not have made a complete 360° rotation.

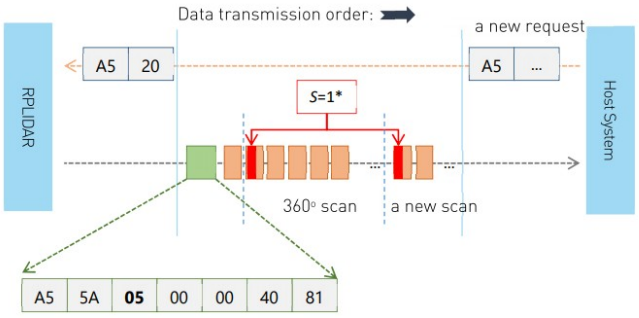


Figure 35 : changement de valeur de S.

Cabin[0]

distance1[5:0]+dθ$_1$[5:4]+distance1[13:6]=0x0000

This give us a distance of 0, or according to the documentation:

Field Definition	Description	Examples/Notes
distance1 distance2	The distance data for the first and second measurement sampling. The unit is millimeter(mm) When the value is 0, the matched sampling point is invalid.	The first sampling time is before the second.
$d\theta_1$ $d\theta_2$	The angular compensation value for the first and second measurement sampling. It uses fix point number in q3 format and the unit is degree. The top digit is sign bit.	Please refer to the following sections for how to calculate the included angle value of every measurement sampling point.

We see that if the distance is zero then the point is invalid, which is the case but it is similar to the behavior of the standard mode that started with “0x01”….

In addition, we note that there are 2 distances per cabin, but we have 16 cabins. This means that 32 points are processed per packet. Thus, the LiDAR makes 1 angle measurement for 32 points and uses the formula to find the angles of the 32 points.

Going back to the received bytes, in addition to the distance, the LiDAR returns dθ1 which is the angular compensation. This is used by LiDAR to compensate errors during its rotation. Indeed, when the LiDAR emits an IR ray at a certain angle θi, the motor continues to rotate up to a certain angle θi+1 . Also, by receiving the emitted ray it must be able to find out at which angle the measurement was made, so he updates dθ, to know how much he must substract to the current angle in order to find the angle associated to the distance received.

Here, this compensation is 0°. And it’s the same compensation for the second distance.

So there is 32 angular compensation per packet, so to come back to the formula, the angle «k» corresponds to the measurement number in the packet (so k is maximum 32). This is why we multiply *AngleDiff* by k/32 because the more we advance in the measurements and the more the gap between the starting angle of packet 1 and packet 2 is large. Thus, we add this to the starting angle of the first packet to converge to the starting angle of the next packet and subtract the angular compensation.