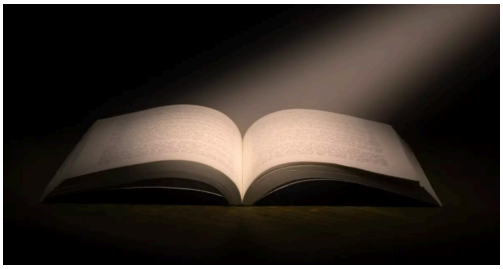
DSHOT - the missing Handbook

Feb 27, 2021 • Chris Landa



DSHOT - **D**igital **shot**, is a very popular protocol for flight-controller to ESC communication. In the quadcopter hobby it is nowadays pretty much the standard. The protocol is used to send the target throttle value from the flight-controller to the ESC which in turn interprets it and drives the motor accordingly.

This article has been translated into Chinese by Hugo Chiang.

This is a **compilation of DSHOT and bidirectional DSHOT implementation details**. When I first started looking into ESC firmware, I had a hard time finding all the needed DSHOT related information - at least not in one place, that is why I decided to create this article. I hope this can be useful for other developers too. My sources are quoted on the bottom, if something is not clear from my writing, they might help you out. If you have any feedback or you found a mistake, please feel free to leave a comment.

TABLE OF CONTENT

History

Before DSHOT there were analog protocols, most commonly known: **PWM**, although there are others like Oneshot and Multishot. Multishot is still quite popular within quadcopter pilots.

A **digital protocol has a couple of huge benefits** in comparison to the analog protocols:

- **Error checking**: a checksum allows the ESC to confirm that the data is truly what has been sent by the flight controller and there was no interference (at least to a certain degree)
- **Higher resolution**: in case of DHSOT 2000 steps of throttle resolution
- **No oscillator drift** and thus no need for calibration
- **Two way communication on one wire**

But, everything has two sides, and so do digital protocols. The downsides are, that the **digital protocols are not the fastest** since they carry an overhead like - in the case of DSHOT - the CRC, which adds reliability but also increases the duration of a frame and thus data needed to be transmitted. Also frames are always of a fixed length, no matter if you are going full throttle or no throttle - wheres the length of a pulse is shorter with analog when the throttle value is smaller.

> *Multishot has a maximum frame duration of 25µs at full throttle and is still more than twice as fast as DSHOT 300 with a constant frame duration of 53.28µs.*

Supported Hardware

DSHOT is supported on all BLHELI_S, BLEHLI_32 and KISS ESC's. One limitation are older BLHELI_S ESC's with EFM8BB1 MCU: Only DSHOT 150 and DSHOT 300 are supported on those, but this should still be good enough for 99% of use cases.

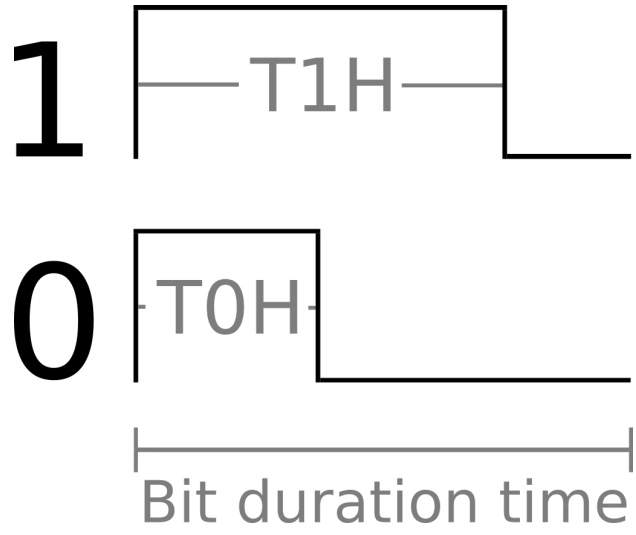
No extra settings on the ESC's are needed - they automatically detect by which protocol they are driven and act accordingly. Some firmware might only support a certain subset of protocols though, so be aware of that. Bluejay for example only supports DSHOT in all its variations, but none of the analog protocols.

Frame Structure

Every digital protocol has a structure, also called a frame. It defines which information is at which position in the data stream. And the frame structure of DSHOT is pretty straight forward:

- **11 bit throttle**: 2048 possible values. 0 is reserved for disarmed. 1-47 are reserved for special commands. Leaving 48 to 2047 (2000 steps) for the actual throttle value

The interesting part is, that 1 and 0 in the DSHOT frame are distinguished by their high time. This means that every bit has a certain (constant) length, and the length of the high part of the bit dictates if a 1 or 0 is being received.



This has two benefits:

1. Every frame has exactly the same, easy to calculate duration: **16 x (bit period time)**
2. The measurement of a bit can always be triggered on a rising flank and stopped on a falling flank (or the other way around in case of the inverted signal with bidirectional DSHOT)

In DSHOT the **high time for a 1 is always double that of a 0**. The actual frame duration, bit period time and frame length depend on DSHOT version:

DSHOT	Bitrate	T1H	T0H	Bit (µs)	Frame (µs)
150	150kbit/s	5.00	2.50	6.67	106.72
300	300kbit/s	2.50	1.25	3.33	53.28
600	600kbit/s	1.25	0.625	1.67	26.72
1200	1200kbit/s	0.625	0.313	0.83	13.28

T1H is the duration in µs for which the signal needs to be high in order to be counted as a 1. **T0H** is the duration in µs for which the signal needs to be high in order to be counted as a 0.

Special commands

As mentioned in the previous section, the throttle values 0-47 are reserved for special commands:

SPECIAL COMMANDS

Nr.	Command	Remark
0	DSHOT_CMD_MOTOR_STOP	Currently not implemented
1	DSHOT_CMD_BEEP1	Wait at least length of beep (260ms) before next command
2	DSHOT_CMD_BEEP2	Wait at least length of beep (260ms) before next command

4	DSHOT_CMD_BEEP4	Wait at least length of beep (260ms) before next command
5	DSHOT_CMD_BEEP5	Wait at least length of beep (260ms) before next command
6	DSHOT_CMD_ESC_INFO	Wait at least 12ms before next command
7	DSHOT_CMD_SPIN_DIRECTION_1	Need 6x
8	DSHOT_CMD_SPIN_DIRECTION_2	Need 6x
9	DSHOT_CMD_3D_MODE_OFF	Need 6x
10	DSHOT_CMD_3D_MODE_ON	Need 6x
11	DSHOT_CMD_SETTINGS_REQUEST	Currently not implemented
12	DSHOT_CMD_SAVE_SETTINGS	Need 6x, wait at least 35ms before next command
13	DSHOT_EXTENDED_TELEMETRY_ENABLE	Need 6x (only on EDT enabed firmware)
14	DSHOT_EXTENDED_TELEMETRY_DISABLE	Need 6x (only on EDT enabed firmware)
15	-	Not yet assigned
16	-	Not yet assigned
17	-	Not yet assigned
18-	-	Not yet assigned
19	-	Not yet assigned
20	DSHOT_CMD_SPIN_DIRECTION_NORMAL	Need 6x
21	DSHOT_CMD_SPIN_DIRECTION_REVERSED	Need 6x
22	DSHOT_CMD_LED0_ON	-
23	DSHOT_CMD_LED1_ON	-
24	DSHOT_CMD_LED2_ON	-
25	DSHOT_CMD_LED3_ON	-
26	DSHOT_CMD_LED0_OFF	-
27	DSHOT_CMD_LED1_OFF	-
28	DSHOT_CMD_LED2_OFF	-
29	DSHOT_CMD_LED3_OFF	-
30	Audio_Stream mode on/Off	Currently not implemented

32	DSHOT_CMD_SIGNAL_LINE_TELEMETRY_DISABLE	Need 6x. Disables commands 42 to 47
33	DSHOT_CMD_SIGNAL_LINE_TELEMETRY_ENABLE	Need 6x. Enables commands 42 to 47
34	DSHOT_CMD_SIGNAL_LINE_CONTINUOUS_ERPM_TELEMETRY	Need 6x. Enables commands 42 to 47 and sends erpm if normal Dshot frame
35	DSHOT_CMD_SIGNAL_LINE_CONTINUOUS_ERPM_PERIOD_TELEMETRY	Need 6x. Enables commands 42 to 47 and sends erpm period if normal Dshot frame
36	-	Not yet assigned
37	-	Not yet assigned
38	-	Not yet assigned
39	-	Not yet assigned
40	-	Not yet assigned
41	-	Not yet assigned
42	DSHOT_CMD_SIGNAL_LINE_TEMPERATURE_TELEMETRY	1°C per LSB
43	DSHOT_CMD_SIGNAL_LINE_VOLTAGE_TELEMETRY	10mV per LSB, 40.95V max
44	DSHOT_CMD_SIGNAL_LINE_CURRENT_TELEMETRY	100mA per LSB, 409.5A max
45	DSHOT_CMD_SIGNAL_LINE_CONSUMPTION_TELEMETRY	10mAh per LSB, 40.95Ah max
46	DSHOT_CMD_SIGNAL_LINE_ERPM_TELEMETRY	100erpm per LSB, 409500erpm max
47	DSHOT_CMD_SIGNAL_LINE_ERPM_PERIOD_TELEMETRY	16us per LSB, 65520us max TBD

Commands 0-36 are only executed when motors are stopped. Some commands need to be sent multiple times in order for the ESC to act on it - those are marked with *Needs nx* - where n is the amount of times the command has to be sent in order for the ESC to act upon it.

For sake of completeness here is the frame structure for a ESC_INFO response for BLHELI_32:

BLHELI_32 ESC INFO FRAME

Calculating the CRC

The checksum is calculated over the throttle value and the telemetry bit, so the "first" 12 bits our *value* in the following example:

```
crc = (value ^ (value >> 4) ^ (value >> 8)) & 0x0F;
```

Let's assume we are sending a throttle value of **1046** - which is exactly half throttle and the telemetry bit is not set:

```
value  = 100000101100
(>>4)  = 000010000010 # right shift value by 4
(^)    = 100010101110 # XOR with value
```

So the two bytes transmitted from flight-controller to ESC would be:

```
1000001011000110
```

We would put this signal on the wire to transmit the DSHOT frame:

The green part are the throttle bits, blue is the telemetry bit and yellow the CRC checksum.

Arming sequence

The arming sequenc might differ from implementation to implementation, but on most firmwares a 0 command is expected for a certain amount of time - 300ms on Bluejay for example - before the ESC goes into a state where it will accept actual throttle values.

Why is frame length important?

The frame length is important because it indicates how fast the ESC can be updated. The shorter the frame length, the more often a frame can be sent per second. In other words the higher the bitrate, the more data we can send per second.

This is then only limited by the loop speed of the flight-controller. Or the other way around, as we will see.

> *Let'shave a look at DSHOT 300: A frame length of 106.72µs allows us to theoretically send 18768 full frames per second. Which results in a maximum frequency of around 18kHz.*

From this example **we can conclude that with a PID loop frequency of 8kHz we can't exhaust DSHOT 300**, so there is no real reason to run DSHOT600 - at least if your PID loop frequency is 8kHz or less.

But **this is actually not the whole truth**, since the flight controller spaces out the frames and locks it to the PID loop frequency. DSHOT frame generation thus always runs at PID loop rate - this on the other hand means, that if you are running really high PID loop frequencies, you also need to run a high DSHOT version.

> *Should you for example run a 32kHz loop, the flight controller will send DHSOT frames every 31.25µs - meaning you have to run at least DSHOT600 in order to keep up.*

What is ESC Telemetry?

In the section about Frames I mentioned a telemetry bit. The flight controller uses this bit to request telemetry information from the ESC.

Telemetry information can be different things, for example the temperature of the ESC, or the eRPM with which the motor is spinning, current draw and voltage.

> ***CAUTION**: Keep in mind that ESC telemetry is not bidirectional DSHOT and the communication is way too slow for RPM filtering to work properly.*

Hardware compatibility

ESC telemetry requires a **separate wire** to transmit information back to the flight-controller. It is generally only available on **KISS and BLHELI_32 ESC's**. The wire to the flight-controller can be shared between multiple ESC's and is connected to the TX pin (for half duplex communication) of an otherwise unused UART.

Which telemetry data is there?

As mentioned in the section above, bits 1-47 are reserved for special commands, some of which are used to request telemetry. Of those commands, 42-47 are related to telemetry - please reference the table to see which telemetry data you can query.

Transmission

via a single line back to the flight controller.

The frame size is a whopping 10 byte - 80 bit and is transmitted with a baudrate of 115200.

All telemetry data is transmitted in this frame. I do not want to go into further detail about ESC telemetry since it is not really part of DSHOT. Detailed specifications can be found in an rcGroups thread.

> *This way of querying is pretty much outdated and too slow to do anything meaningful - except if you are interested in the current draw directly at the ESC.*

Bidirectional DSHOT

Bidirectional DSHOT is available in BLHELI_32 and on BLHELI_S when using 3rd party firmware as described in my article about RPM filtering, where I compare different firmware options.

Biderictional DSHOT is also known as **inverted DSHOT**, because the signal level is inverted, so 1 is low and a 0 is high. This is done in order to let the ESC know, that we are operating in bidirectional mode and that it should be sending back eRPM telemetry packages.

> *Bidirectional DSHOT only works with DSHOT 300 and up.*

Calculating the Checksum

The calculation of the checksum is basically the same, just before the last step the values are inverted:

```
crc = (~(value ^ (value >> 4) ^ (value >> 8))) & 0x0F;
```

With the same values as for the regular DSHOT frame:

```
value  = 100000101100
(>>4)  = 000010000010 # right shift value by 4
(^)    = 100010101110 # XOR with value
(>>8)  = 000000001000 # right shift value by 8
(^)    = 100010100110 # XOR with previous XOR
(~)    = 011101011001 # Invert
(0x0F) = 000000001111 # Mask 0x0F
(&)    = 000000001001 # CRC
```

Bidirectional DSHOT Frame (from flight-controller)

The frame sent from the flight controller to the ESC has exactly the same structure, just the signal is inverted. The two bytes transmitted from flight-controller to ESC would be:

```
1000001011001001
```

On the wire, the signal would look like this:

need to keep this in mind, especially when running higher PID frequencies.

Although in bidirectional DHSOT mode, eRPM are always returned, other telemetry information can still be requested, but is then sent via a separate wire.

Once the flight controller sends its frame, it switches to receive mode and waits for the eRPM frame to be returned from the ESC. The same thing is happening on the ESC - when the flight-controller is sending, the ESC is listening and the other way around.

After sending a frame there is a 30µs break to switch the line, DMA, and timers in order for a frame to be received. This break is independent of DSHOT frequency.

eRPM Telemetry Frame (from ESC)

The eRPM telemetry frame sent by the ESC in bidirectional DSHOT mode is again a 16 bit value, so the same size as the received frame, but the structure is different:

- **12 bit**: eRPM Data
- **4 bit**: CRC

The encoding of the eRPM data is not as straight forward as the one of the throttle frame:

- **3 bit**: Amount that the following values need to be left shifted in order to get the periods in µs
- **9 bit**: Period base

```
eeemmmmmmmmmcccc
```

The CRC is calculated exactly as it is with uninverted DSHOT, it is also sent back to the flight-controller uninverted.

Extended DShot Telemetry (EDT)

Extended DShot telemetry or EDT is a relative late addition to the DShot "standard". It allows transmission of additional telemetry within the eRPM frame, meaning no additional wire is needed to transmit additionsl telemetry from the ESC to the flight-controller.

> *This needs to be supported by both sides, ESC and flight-controller. Bleujay, BLHeli_32, AM32 all support this feature in their latest version.*

This is done by a special means of encoding: The eRPM telemetry is a bit redundant, if we just look at the data, ignoring the CRC:

```
eee mmmmmmmmm
123 123456789
```

if we have a closer look, we can see that the same eRPM value can (under certain circcumstances) be coded in different ways:

```
000 000000010 = 000000010
001 000000001 = 000000010

000 000000100 = 000000100
001 000000010 = 000000100

000 010101010 = 010101010
001 001010101 = 010101010

110 000000010 = 010000000
111 000000001 = 010000000

000 011111110 = 011111110
```

value once and decrementing the exponent. If we always encode like this, there will never be eRPM values that look like so:

```
ee1 0mmmmmmmm
ee0 0mmmmmmmm
```

And this we can use to encode additional data. If we see a frame that starts like so:

```
eee 0mmmmmmmm
```

we know that it is not a regular eRPM frame but instead an extended DShot telemetry frame and we will interpret the package like so:

```
pppp mmmmmmmm
```

First four bits are telemetry type, last 8 bits are value. Since the last bit of telemetry type is alwys zero, we can encode the following telemetry types with an 8 bit value (0-255):

Type	Description
0x02	Temperatur in C
0x04	Voltage: 0.25V per step
0x06	Current in Amp
0x08	Debug value 1
0x0A	Debug value 2
0x0C	Debug value 3
0x0E	State/Event

Which of those types are actually transmitted depends on the firmware and the ESCs capability. Temperature is supported on basically all of them since most MCUs have an internal temperature sensor anywa. BLHeli_32 and AM32 also support voltage and current.

Debug values can be anything the developers want it to be and allow for easy run time debugging.

> *The EDT frame is encoded and transmitted as any other eRPM frame. The frequency in which the telemetry frames are sent is basically up to the implementation. It is just important to note that they should not be sent too often as to not interfere with the RPM filtering.*

eRPM Transmission

But there is a twist, it is not actually this value that is being sent back to the flight controller. Instead GCR encoding is applied and the 16 bit value is mapped to a 20 bit value by mapping the nibbles (groups of 4 bit) according to the following table:

Nibble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Mapped	19	1B	12	13	1D	15	16	17	1A	09	0A	0B	1E	0D	0E	0F

If we take the following example value:

```
16 bit:  1000001011000110
Nibbles:  1000  0010  1100  0110
```

Now we mapped our 16 bit value to 20 bit, but this is not ready for transmission yet. A 21 bit needs to be added and the original bits are transformed following this rules:

We map the GCR to a 21 bit value, this new value starts with a 0 and the rest of the bits is set by the following two rules:

1. If the current bit in GCR data is a 1: The current new bit is the inversion of the last new bit
2. If the current bit in GCR data is a 0: The current new bit is the same as the last new bit

This is best explained with a short example. Lets assume we have the GCR value of `01100010`:

```
GCR:  01100010
New: 0          # We start out with adding a 0 bit (the 25th bit in the real frame)
     00         # 1 bit of GCR is 0 => Rule 1: new bit is 0, because the last was 0
     001        # 2 bit of GCR is 1 => Rule 2: new bit is 1 after inverting the last bit
     0010       # 3 bit of GCR is 1 => Rule 2: new bit is 0 after inverting the last bit
     00100      # 4 bit of GCR is 0 => Rule 2: new bit is 0, because the last was 0
     001000     # 5 bit of GCR is 0 => Rule 2: new bit is 0, because the last was 0
     0010000    # 6 bit of GCR is 0 => Rule 2: new bit is 0, because the last was 0
     00100001   # 7 bit of GCR is 1 => Rule 2: new bit is 1 after inverting the last bit
     001000011  # 8 bit of GCR is 0 => Rule 2: new bit is 1, because the last was 1
```

Let's look at a real, 20 bit GCR value - the worst case scenario, where each bit is different

```
GCR:  10101010101010101010
New: 01100110011001100110
```

When we put this value on the wire, we only have half the transitions (switching from high to low) than we would have by sending the original GCR value.

So instead of sending this:

We send this:

This value is then sent uninverted at a bitrate of 5/4 x current DSHOT bitrate. So on DSHOT600 the 21 bits are sent with a bitrate of 750kbit/s.

You might now ask yourself: **WHY?** And that is a good question. It turns out that GCR is an excellent transmission format, very robust to jitter. Tests during implementation have shown that the error rate of the eRPM packets is significantly lower when using GCR in comparison to sending DSHOT frames back to the flight-controller.

Decoding eRPM frame (on flight-controller)

On the receiving end it is also pretty simple to decode the frame:

```
gcr = (value ^ (value >> 1));
```

The value only needs to be XOR'd with itself after shifting it to the right once:

```
value = 01100110011001100110
(>>1) = 00110011001100110011 # right shift value by 1
(^)   = 01010101010101010101 # GCR
```

Sources

- DSHOT example FPGA plugin

- Bidirectional DSHOT ESP32
- Betaflight bidirectional DSHOT PR
- CRC DSHOT vs. bidirectional DSHOT
- Betaflight EDT PR