

# PCM1024Z format: What's Known?

W.Pasman 11/11/3

## Introduction

This report documents how the Futaba PCM1024Z data format probably looks like. I combined the autopilot [autopilot03], the smartpropo code [smartpropo02] and the comments I got from several people on RunRyder [RunRyder03], into what is supposed to be the Futaba PCM 1024Z format. I haven't measured anything, this is purely based on 'paper work'. Several details are still missing, this document gives a starting point from where further investigation can start. We go through the protocol bottom-up, starting at bit level and ending at multi-frame level.

## Bits

At the lowest radio level, we have bits. Every bit takes  $150\mu\text{s}$  (micro seconds). A low bit means the f-b frequency is active, and a high bit to be the f+b frequency to be active. f is the base band (35MHz, 40MHz, 72MHz whatever you have), and  $b=3\text{kHz}$ .

The smartpropo software measures the length between flanks in the data where input changes from 0 to 1, and calculates from that how much bits passed. It relies on 44.1kHz audio samples, so every bit is oversampled 6.623 times.

The autopilot project uses an timed interrupt, to measure the incoming bits at pre-set times. The interrupt measures the bit, shifts it into a register until enough bits are gathered, and then sets a semaphore. The rest of the software busy-waits until the semaphore sets, and then processes the register<sup>1</sup>.

## *Inverting the Frame Bits*

All the bits in the radio stream are frequently inverted, probably to avoid problems with frequency skewing of phased locked loops in the transmitter [Patent 5,799,045]. The sync pulse is also inverted: a normal sync pulse is high, the inverted pulse is low. This inversion toggles every two frames, so two normal frames are transmitted, followed by two inverted frames. I guess that this inversion is done in the radio module as suggested in mentioned patent, following a sync pulse.

## pcm byte

Those bits are grouped in 10-bit words. pcm\_byte they are called in the autopilot project. Those pcm\_bytes are 10-bit codes representing 6 data bits. So for every 10 bits received in the radio we have 6 databits left for further processing. This is called block coding, or 6B10B at some places. autopilot and smartpropo have different conversion routines doing the same. Every pcm\_byte takes  $10 \cdot 150 = 1500\mu\text{s} = 1.5\text{ms}$  to transmit.

---

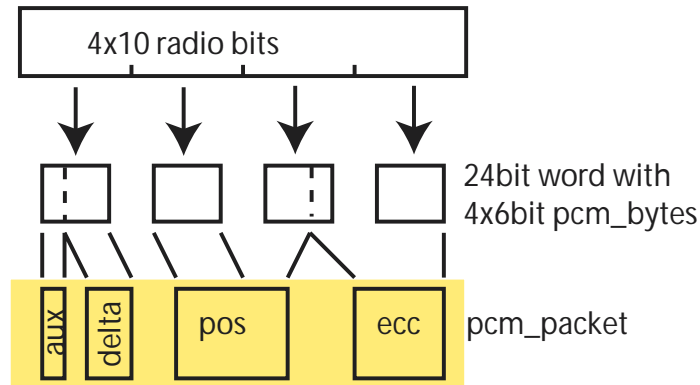
<sup>1</sup> I'm extremely amazed by this, their CPU is occupied full-time with decoding the format and no other program can simultaneously run on their CPU. I thought busy-waiting is a prehistoric solution.

**Table 1.** 6B10B block code. Every 10 radio bits (left column) are representing 6 data bits.

10-bit radio word	6-bit data (decimal)	6-bit data (hexadecimal)
111111000	0	00
111110011	1	01
1111100011	2	02
1111100111	3	03
1111000111	4	04
1111001111	5	05
1110001111	6	06
1110011111	7	07
0011111111	8	08
0001111111	9	09
0000111111	10	0A
1100111111	11	0B
1100011111	12	0C
1100001111	13	0D
1110000111	14	0E
1111000011	15	0F
0011111100	16	10
0011110011	17	11
0011100111	18	12
0011001111	19	13
1111001100	20	14
1110011100	21	15
1100111100	22	16
1100110011	23	17
111110000	24	18
1111100000	25	19
1110000011	26	1A
1100000111	27	1B
1100011100	28	1C
1110011000	29	1D
1110001100	30	1E
1100111000	31	1F
0011000111	32	20
0001110011	33	21
0001100111	34	22
0011100011	35	23
0011111000	36	24
0001111100	37	25
0000011111	38	26
0000001111	39	27
0011001100	40	28
0011000011	41	29
0001100011	42	2A
0000110011	43	2B
1100110000	44	2C
1100011000	45	2D
1100001100	46	2E
1100000011	47	2F
0000111100	48	30
0001111000	49	31
0011110000	50	32
0011100000	51	33
0011000000	52	34
1111000000	53	35
1110000000	54	36
1100000000	55	37
0001100000	56	38
0001110000	57	39
0000110000	58	3A
0000111000	59	3B
0000011000	60	3C
0000011100	61	3D
0000001100	62	3E
0000000111	63	3F

## PCM Packet

Four of these pcm bytes are grouped into a  $4 \times 6 = 24$  bit word, with the first incoming pcm\_byte at the most significant position. Then the bits in this word are re-grouped to form a 2-bit pcm\_aux, 4-bit pcm\_delta, 10 bit pcm\_position and 8 bit pcm\_ecc field. Together these four fields are called pcm\_packet in the autopilot project. Figure 1 shows the steps so far. A pcm packet takes  $40 \times 150 \mu s = 6 \text{ms}$  to transmit.



**Figure 1.** conversion from 40 radio bits to a 24bit pcm packet.

### ***pcm position***

The position information for a channel straightly represents a servo pulse signal. A 0 represents a servo pulse of  $920 \mu\text{sec}$  and a 1023 corresponds to a servo pulse of  $2120 \mu\text{sec}$  (e.g., [Patent 5,799,045]). That gives  $1.2 \text{msec}$  for 1024 positions, or  $1.1718 \mu\text{s}$  per step.

### ***pcm\_ecc***

Sekiriki (the developer of smartpropo) mentions on his board the details of the ecc. The ecc holds the XOR of 8-bit numbers associated with the first 16 bits. The numbers associated with the bits are shown in Figure 2. To calculate the ECC, we start with  $\text{ecc} = 0$ , and run through the first 16 bits in the pcm\_bytes (so, up to the 6th bit in byte 3, as each pcm\_byte holds 6 bits). For each '1' bit, the ecc xor-ed with the associated number from Figure 2. Appendix A explains how this code can be used by the receiver.

6B, D6, C7, E5, A1, 29, 52, A4, 23, 46, 8C, 73, E6, A7, 25, 4A

**Figure 2.** 16 8-bit numbers (hexadecimal) associated with the first bits in the pcm packet.

### ***pcm delta***

The 4 bit delta field tells how to calculate the new position for a channel given the old position. According to the autopilot project we have

$$\text{new position} = \text{old position} + \text{pcm\_delta} - 8$$

However this doesn't match what is suggested in patent 4,916,446, and it is also confirmed by Angelos [Runryder03] that this should be

```
new position=old position + delta_table[pcm_delta]
```

The contents of the delta\_table (16 signed integers) are not publicly available.

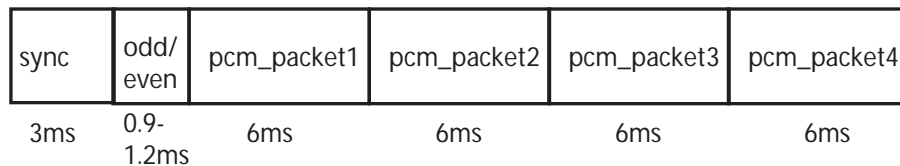
### ***pcm aux***

I have not seen much information yet on the contents of the two auxiliary bits. They may contain failsafe information, failsafe behaviour when the battery goes low, channel 9 (and maybe even more channels?).

One thing that almost surely resides here is the position of channel 9. Channel 9 is a switch channel and thus needs only 1 bit. Sekiriki mentions it's in the "18th bit of every frame". However that can't be the case because bit 18 falls within the pcm\_ecc block. Counting the other way round, starting at the lowest ecc bit, we would get midway the pcm\_pos field. My guess is that he miscounted and that channel 9 is in the second bit of the second pcm\_aux block. Channel 10 is probably another bit in the aux fields.

## **Frames**

Pcm packets are grouped into clusters of 4, called frames. A frame holds all the information to update all the channels. There are two types of frames: an odd and an even frame. Every frame is preceded by special bits called the sync, which is meant to allow the radio level to synchronise and to recognise the start of data. After that follows the frame odd/even code, and then come the four pcm packets. Figure 4 shows this.



**Figure 4.** Global structure of a frame. It starts with sync pulses for the radio level, then comes an odd/even frame indicator, and then follows the data.

### ***Sync***

A sync pulse usually lasts 3ms, and during that time only 1 'high' bit is transmitted. Smartpropo looks for a sync pulse of 2700µs (18 bits), which looks a bit too rigid and might cause loss of frames (maybe intended?). The autopilot code wait till a sync pulse of at least 2500µs comes by.

### ***Odd/Even Code***

Directly after the sync follows the odd/even frame indication.

According to the smartpropo code, this is coded with six bits (not a pcm\_byte, but straight 150ms pulses at radio level). For an odd frame the bits are 000011\*\* and for an even frame 000000<sup>2</sup>. The \*\* indicate 2 bits that are ignored in the smartpropo system.

In the autopilot project, these odd/even code bits are skipped by setting a timer, and the extra 2 bits for the odd frames are just accounted for by adding 300µs to the timer. On top of that they have to correct for normal and inverted frame bits. Most likely these extra

---

<sup>2</sup> This may be the other way round, because smartpropo and autopilot don't agree on the actual contents of the frame.

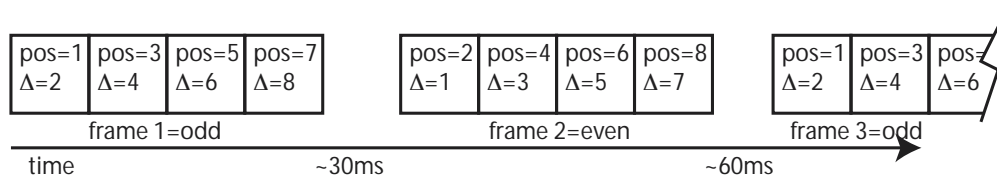
corrections are caused by their rigid fixed-time sampling using an interrupt where smartpropo can accept a bit of variation on the timings.

Note: Angelos [Runryder03] talks about "failsafe frames", so there might be a third header indicating this.

### Odd and Even Frame

We can number the frames in the order they are transmitted. This gives us odd-numbered and even-numbered frames. In every frame we have four pcm\_packets <pcm\_pos,pcm\_delta,...>, each targeting two channels {target for pcm\_pos, target for pcm\_delta}. The target varies, depending on whether a frame is odd or even.

In an odd frame, the target channels are {1,2} {3,4} {5,6} and {7,8} for the subsequent packets. In even frames, the target channels are {2,1} {4,3} {6,5} and {8,7}. Stated otherwise, the pcm\_pos and pcm\_delta is alternatingly used by one or the other channel. Figure 5 shows this.



**Figure 5.** In odd frames, even channels use the pcm\_pos and odd channels the pcm\_delta field of the pcm\_packets. In even frames this is the other way round.

Above configuration is according to autopilot code; according to the smartpropo code the order is<sup>3</sup> {3,2}{4,1}{5,4} for the odd frame and {2,3} {1,4} {4,5} for the even frame. He doesn't mention channel 6 and 7.

Angelos [Runryder03] suggests yet another order: {5,6} {1,2} {3,4} {7,8}. He doesn't talk about odd and even frames.

In short, there is big confusion about the order of the frames. Maybe there is some software setting in the radio that can change the order? This may make sense for some applications. But then, this information has to be available somewhere in the packets as well. Alternatively, it could be that odd and even frames have a different order.

### Total frame length, extra bits?

Apart from the sync and odd/even information, a frame thus lasts 28ms, including the odd/even frame info this would be 28.9 or 29.2ms, and including the sync we get a grand total of 31.6ms and 31.9ms depending on the frame type. Note that this is much longer than PPM frames that last only 22.5ms.

Angelos [Runryder03] mentions 28.2 or 28.8ms frame time, depending on the 'sync type which defined the data it carries'. From the notes he gives on latency I deduce that we agree on the 'start of the frame', namely after the 6- or 8-bit frame type pulse. That means that he suggests that there two to five extra bits after the end of the last pcm frame that have not been reported in literature.

<sup>3</sup> I'm making an educated guess where Sekiriki thinks the delta code is

## References

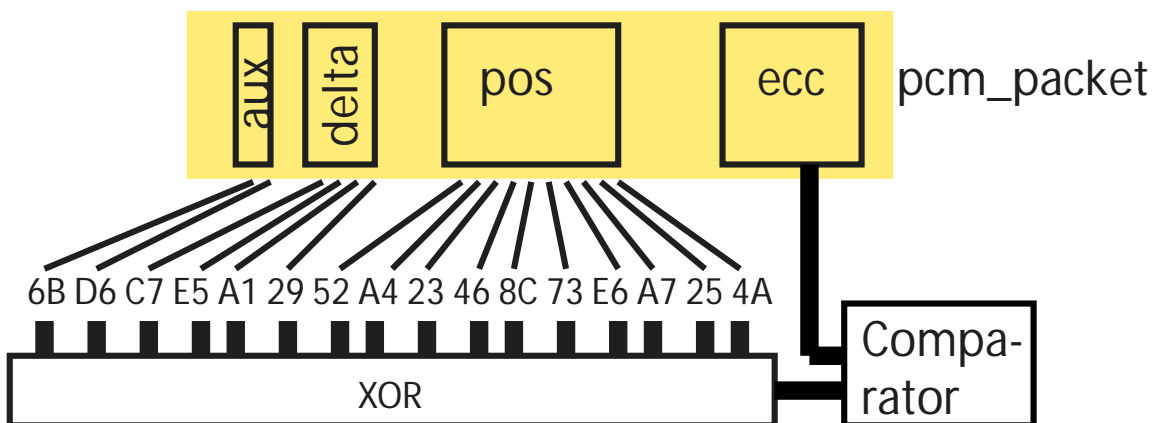
- [autopilot03] Hudson, T. (2003). Autopilot: Do it yourself UAV.  
<http://sourceforge.net/projects/autopilot> and  
<http://autopilot.sourceforge.net>.
- [smartpropo02] Sekiriki (2002). SmartPropo: The RC to PC Audio Interface.  
<http://www.sekiriki.jp/smartpropo/index.html>
- [Runryder03] Decoding Futaba PCM 1024Z.  
<http://www.runryder.com/helicopter/t71094p1>
- [Pat. 5,799,045] Sakuma (1998). patent 5,799,045: PLL-mode radiofrequency module.  
Available Internet: <http://www.uspto.gov/>.
- [Pat. 4,916,446] Yamamoto (1990). Patent 4,916,446: Remote Control device.  
<http://www.uspto.gov/>.

## Appendix A: error correction

In the receiver, the ecc can be used to check whether all bits were transmitted correctly (Figure 3). To do this, ecc2 is calculated by re-doing the calculation using the received bits. If ecc (the one received) equals ecc2, all bits probably are okay.

If 1 bit is wrong, we can easily find out which one by xor-ing ecc and ecc2, and look up the resulting number in Figure 2. The bit having this number is wrong and has to be inverted. For instance if ecc XOR ecc2 = E5, the second bit of the pcm\_delta field is wrong.

The numbers are constructed such that if two bits are wrong, never a number already in Figure 2 is generated, avoiding malicious 'correction' by flipping only 1 bit. If two bits are corrupted, we can't do much, because we have  $16 \cdot 15 / 2 = 120$  possible combinations of two bits in 16 bits being wrong, while the pcm\_ecc can represent only 256 different numbers. In such a case, the chances would be too high that the ecc itself was mangled during transmission. Furthermore, it can not unique be determined in all cases which bits went wrong anymore if two bits are corrupted.



**Figure 3.** In the receiver, the ecc that would follow from the first 16 bits from a pcm packet is compared with the ecc as it was calculated in the transmitter. Results should be the same if no bits are damaged in the transmission.

## **Appendix B: Receiver oddities**

This section contains some remarks that are not part of the protocol but nevertheless are interesting for the latency aspects of the total system. In the receiver the servo pulses may not be generated in incoming order. Angelos [Runryder03] reports the timings for generating the servo pulses relative to the start of their respective frames<sup>4</sup> (Table 2).

channels	pulse start relative to frame start (ms)
1 and 2	9.480
3 and 4	7.120
5 and 6	7.120
7 and 8	8.4

Table 2. Servo pulse delays with respect to

The frame transmission time of 31.6ms is long enough to generate two servo pulses per frame, and Angelos [Runryder03] indicates that this is indeed the case. He also knows that the second pulse is always identical to the first one, and always generated 14.16ms after the first pulse. This might give some interesting problems to the servos, as smooth interpolation would be hampered by such 'fake' pulses.

---

<sup>4</sup> Angelos talks about 'sub frames' and 'relative to first bit of the frame'. As far as I can see he means relative to what we call pcm\_packets.