PCM1024Z Format: Reverse Engineered

W.Pasman, F. Goddeeris, 6/1/4

Introduction

This report documents Futaba PCM1024Z data format. This report is the combined result of previous known knowledge from the internet [Pasman03] and reverse engineering of which multiple people [Runryder03]. Source codes of PCM decoders [Sekiriki02b], [Autopilot02] and a few important details [Sekiriki02] were also an important source of knowledge.

We start with a short overview of the available channels. This is the data that has to be transferred to the receiver. Then we discuss how PCM1024 represents this data. We distinguish three steps in the construction of the PCM1024 packets. First, there is the source coding, which describes how the channels are encoded into data packets, how the source data is compressed, what extra bits are added etc. Next, there is the adding of error protection that allows to detect and maybe even correct errors that occurred during transmission of the data to the receiver. Finally there is the channel coding, which describes how the data is modulated and adapted to fit it on top of a radio wave carrier.

The appendices discuss a few issues that are not central to the PCM1024Z format, but are relevant to understand our results.

Channels

This section gives a short overview of the channels that have to be transmitted in the 1024Z format. We distinguish the non-real time and the real-time channels.

Not all transmitters use all channels, and sometimes channels are controlled in unexpected ways. Appendix B describes this.

Non-Realtime Channels

The non-realtime channels are channels that only need very infrequent update. They are transitted four to eight seconds after turning on the transmitter, and after that once per minute. Additionally they are sometimes transmitted immediately after changing (see Appendix B).

As far as we know only failsafe data is transmitted on the non-realtime channel.

The receiver can enter a failsafe mode when it detects that control is lost or will be lost soon. Currently there are two such modes: battery failsafe and radio failsafe.

Battery failsafe is a special mode that the receivers goes to whenever the receiver battery voltage goes too low (below 3.8V). When the receiver goes into this mode, the throttle channel (channel 3) moves to a predetermined position. The other channels remain working in real time – as long as the battery permits of course–.

When the receiver has sustained radio reception failures, the receiver will go to the radio failsafe mode. This mode is normally referred to as 'failsafe' but we use 'radio failsafe' here to distinguish it from the battery failsafe mode. The radio failsafe condition is triggered by sustained radio reception failures[1]. When the radio enters radio failsafe mode, the receiver moves channel 1 to 8 to a predetermined position.

[1] The FF9 receiver goes to failsafe after 70 half bad frames. For other receivers this might be different.

Ideally every realtime channel (Table 1) would have a corresponding failsafe channel, but as far as we know only failsafe-channel 1 to 8 exist. Failsafe settings for channel 1 to 8 consist of a "mode" value ('normal' or 'preset') and a "position" value (0-1023). If a failsafe mode for a channel is set to 0 ('normal'), the channel is supposed to hold the last position in case of a battery or radio failsafe condition. The 'position' value is ignored in that case. If the failsafe mode is 1 ('preset'), the channel is supposed to go to the pre-set value as defined in the position value for the failsafe setting.

The Real-Time Channels

The real-time channels are the channels that appear normally on the receiver output. They are updated approximately 35 times per second, and therefore follow your stick and switch movements in 'real time'.
PCM1024 offers place to eight real-time proportional channels and a few switch channels. Proportional channels offer a continuous value that is digitized as an unsigned 10 bits number (so can have values between 0 and 1023, with 512 in the middle), switch channels can only be on or off.
Currently up to two switch channels are provided by the transmitter, one documented and one not documented, but it seems that there is space in the format to accomodate more channels. Table 1 shows the currently known channels and their name in the standard helicopter setup.
The battery failsafe channel is a bit special, related to the receiver 'battery failsafe' mode. The transmitter can reset this mode, which returns the receiver for 30 seconds to non-battery-failsafe mode, allowing temporary regain of throttle control. To do this, the transmitter needs to send a 1 over the BFR channel, it is 0 normally. Appendix B describes when the transmitter sends a battery failsafe reset. The battery failsafe reset channel (BFR) is not documented by Futaba as a separate channel but we found it convenient to do so because we found that the battery failsafe reset mechanism works similar to switch channel 9.

Table 1. channel numbers and usual channel assignment in helicopter mode.

channel number	Helicopter channel assignment	Channel type
1	aileron	proportional
2	elevator	proportional
3	throttle	proportional
4	rudder	proportional
5	gyro sensitivity	proportional
6	pitch	proportional
7	channel 7	proportional
8	channel 8	proportional
BFR	battery failsafe reset	switch
9	channel 9	switch
10	channel 10	switch

The calculation of the channel values depends on the settings of the transmitter: the stick positions, switch positions, mixer settings, etc. The Futaba transmitter manuals describe how the values for the channels are calculated in the transmitter.

Source coding

This section describes the source coding: how all the normal and failsafe channels are packed into data bits. We start with a discussion of the real-time channel coding, and then discuss how the non-realtime channels are put through.

Real-time channels

The real-time channel data is encoded into datapackets. Every datapacket holds 16 bits of data: two auxiliary bits, a four bit difference from a previous position, and a ten bit position (Figure 1).

Auxiliary		Delta				Position									
A1	A0	D3	D2	D1	D0	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Figure 1. 16 bits in a datapacket.

Two of those datapackets form a packetpair (Figure 2). Aux bit B3 of a packetpair is always 1 in real-time data frames. In packetpairs we use the letter B for aux bits instead of A, to avoid confusion about the bit positions.

Aux		Delta			Position									Aux		Delta			position									
B3	B2													B1	B0													

Figure 2. Two datapackets form a packetpair. B3=1 in realtime frames.

Four of these datapackets, or two packetpairs, are needed to transmit all proportional real-time channel data. Such a block is called a **frame**. The switch channels are transmitted only once in two frames. Figure 3 shows an overview of the packet contents.
The first datapacket in a frame holds the channels 1 and 2, the second channels 3 and 4, the third channels 5 and 6, and the fourth channels 7 and 8. One of the two channels in each datapacket is coded/compressed with a delta code, the other channel is transmitted uncompressed.
Every channel is transmitted alternatingly compressed (into the delta field) and uncompressed (into the position field). In the even numbered frames, channel 1, 3, 5 and 7 go uncompressed, and in the odd numbered frames channel 2, 4, 6 and 8 are uncompressed.
The delta encodes the difference between the current channel value and the last transmitted value. Table 2 shows the actual differences, and the delta code associated with each difference. The last column shows the difference that is probably used by the receiver, assuming it tries to avoid overshooting.
Channel BFR, channel 9 and channel 10 are all coded into aux bit B0, of packetpair 0 and 1 in even frames, and packetpair 2 of odd frames respectively.

Table 2. position differences and the delta code used for those differences.

delta code	difference	jump probably used in receiver
0	-116 .. -1023	-116
1	-88 .. -115	-88
2	-64 .. -87	-64
3	-44 .. -63	-44
4	-28 .. -43	-28
5	-16.. -27	-16
6	-8.. -15	-8
7	-4 .. -7	-4
8	-3 .. 4	0
9	5 .. 8	5
10	9 .. 16	9
11	17 .. 28	17
12	29 .. 44	29
13	45 .. 64	45
14	65 .. 87	65
15	88 .. 1023	88

When injection of non-realtime data (see below) into the frame causes a channel to miss its 'opportunity' to update the absolute position, that channel may need a larger jumps in its subsequent delta slot to catch up. We did encounter some apparently too large delta codes in such situations, so it's not entirely clear what happens here.
Aux bits B2 and B1 apparently are unused in the realtime frames.

frame	packetpair	datapacket	aux A1	aux A0	delta	position
even	0	1	1		ch2	ch1
		2		bfr	ch4	ch3
	1	3	1		ch6	ch5
		4		ch9	ch8	ch7
odd	0	1	1		ch1	ch2
		2		ch10	ch3	ch4
	1	3	1		ch5	ch6
		4			ch7	ch8
even	0	1	1		ch2	ch1
		2		bfr	ch4	ch3
	1	..	..	..	..	..

Figure 3. Channel assignments for the four datapackets in odd and even frames. This shows the usual case holding only realtime channel data. A number of aux bits are not used, those are usually 0.

Non-Realtime Channels

Instead of occupying a full frame, the non-realtime data is injected into the a fraction of the realtime datapackets. The aux bits are used to indicate when, where and which non-realtime data has been injected. In such a case, part of the real-time channel is forced out of the frame, and therefore can not be updated. This causes a short hickup in the reception of the overridden realtime channels.

If transmitted, non-realtime data for channel *N* (*N*=1..8) overrides the realtime position field for channel *N*. Aux bit B3 is set to 0 if non-realtime data is being mixed into a realtime frame. If that is the case, aux bit B2 determines whether the position field in this datapacket is overwritten with non-realtime data (B2=0) or the position in the next datapacket is overwritten (B2=1). The overwritten position value is the failsafe value for the channel that would normally been transmitted. In both cases, the failsafe mode value (FSM) is put in B1.

All other fields are untouched, and contain normal real-time data. Figure 4 shows the possible configurations of the aux bits, and where the non-realtime data goes. Empty fields are untouched, and hold real-time data.

frame	Packet pair	B3	B2	delta	Pos	B1	B0	delta	Pos
even	0	**0**	**0**	ch2	**FSCH1**	**FSM1**	bfr	ch4	ch3
	1	**0**	**0**	ch6	**FSCH5**	**FSM5**	ch9	ch8	ch7
odd	0	**0**	**0**	ch1	**FSCH2**	**FSM2**	ch10	ch3	ch4
	1	**0**	**0**	ch5	**FSCH6**	**FSM6**		ch7	ch8
even	0	**0**	**1**	ch2	ch1	**FSM3**	bfr	ch4	**FSCH3**
	1	**0**	**1**	ch6	ch5	**FSM7**	ch9	ch8	**FSCH7**
odd	0	**0**	**1**	ch1	ch2	**FSM4**	ch10	ch3	**FSCH4**
	1	**0**	**1**	ch5	ch6	**FSM8**		ch7	**FSCH8**

Figure 4. Channel assignments when non-realtime data is transmitted. Bold letters show differences with normal frames. non-realtime data FSCH# overrides one realtime data. In fields with small letters contain normal realtime data as in Figure 3. Typically those four frames are sent immediately after each other, but the order can differ. The order shown here is used by the FF8S.

Note that the repetition of bits B2 and B3 (twice per frame) is strange. Maybe this structure gives the possibility to transmit only four channels, by transmitting only the first packetpair. Bit B0 of odd frames seems unused.

Error Protection

Error protection is done by adding eight extra bits to every datapacket. This gives us a new larger datapacket, which we named pcm_packet. Figure 5 shows the layout of the pcm_packet.

aux		Delta				position										crc							
1	0	3	2	1	0	9	8	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Figure 5. pcm_packet: datapacket with added error detection code (crc).

This eight bit code is a CRC code. Briefly, it is calculated by XOR-ing the values in Table 3 for every bit in the datapacket that is 1. This code can also be represented with CRC polynomial $x^8 + x^6 + x^5 + x^3 + x + 1$. An efficient shift register implementation is possible to detect errors. More details on CRC codes can be found in [Kassam99]. It shows that all single and double errors, any odd number of errors, any error burst of length ≤8 and most larger error bursts can be detected. Furthermore it is possible to correct one-bit errors. However it is hard to detect in practice if only one bit was damaged, so it is unlikely that Futaba uses error correction.

Table 3. XOR values for calculating the CRC.

BIT	Value for XOR (hexadecimal)
A1	6B
A0	D6
D3	C7
D2	E5
D1	A1
D0	29
P9	52
P8	A4
P7	23
P6	46
P5	8C
P4	73
P3	E6
P2	A7
P1	25
P0	4A

Channel coding

The channel coding converts the data bits into streams of bits that can be modulated by the radio. It manipulates the data a bit, to ensure the following properties for the radio signal:

- Lower the high frequency components by eliminating isolated 1's or 0's[2].
- Add odd/even frame data because the receiver needs that
- Add sync pulses to enable the receiver to find the start of the frames
- Add preamble to ensure the sync pulse gets the right length and polarity.

Eliminating isolated 1's or 0's

This is done by splitting the pcm_packets into 4 chunks of 6 bits, and re-coding each chunk of 6 bits with with 10 bits that have this property. This is called 6to10 coding. Table 4 shows the particular codes used in PCM1024.

[2] The goal of this coding seems to avoid high frequency components.

Table 4. 6to10 bit coding. Every 6 databits (left column) are converted into 10 bits that don't have isolated 0's or 1's.

6-bit data (decimal)	6-bit data (hexadecimal)	10-bit radio word
0	00	1111111000
1	01	1111110011
2	02	1111100011
3	03	1111100111
4	04	1111000111
5	05	1111001111
6	06	1110001111
7	07	1110011111
8	08	0011111111
9	09	0001111111
10	0A	0000111111
11	0B	1100111111
12	0C	1100011111
13	0D	1100001111
14	0E	1110000111
15	0F	1111000011
16	10	0011111100
17	11	0011110011
18	12	0011100111
19	13	0011001111
20	14	1111001100
21	15	1110011100
22	16	1100111100
23	17	1100110011
24	18	1111110000
25	19	1111100000
26	1A	1110000011
27	1B	1100000111
28	1C	1100011100
29	1D	1110011000
30	1E	1110001100
31	1F	1100111000
32	20	0011000111
33	21	0001110011
34	22	0001100111
35	23	0011100011
36	24	0011111000
37	25	0001111100
38	26	0000011111
39	27	0000001111
40	28	0011001100
41	29	0011000011
42	2A	0001100011
43	2B	0000110011
44	2C	1100110000
45	2D	1100011000
46	2E	1100001100
47	2F	1100000011
48	30	0000111100
49	31	0001111000
50	32	0011110000
51	33	0011100000
52	34	0011000000
53	35	1111000000
54	36	1110000000
55	37	1100000000
56	38	0001100000
57	39	0001110000
58	3A	0000110000
59	3B	0000111000
60	3C	0000011000
61	3D	0000011100
62	3E	0000001100
63	3F	0000000111

Odd and Even Frame Code

The receiver needs to know whether the frame is odd or even. For some reason Futaba decided not to put this into the aux bits but to put a frame code bluntly (without error checking code) in front of the frame. Table 5 shows the bits that are prepended to odd and even frames. Probably they use different code lengths as an alternative to error checking: bits may get mangled but the length of the code can't be changed by interference.

Table 5. Odd or even frame code.

frame	frame code
even	000011
odd	00000011

Sync Pulse, Preamble

The sync pulse is simply a string of 18 consecutive 1's prepended before the odd/even frame pulse. Note strings with more than 16 the same bits never occur in the 6to10 codes, because all codes start and end always with at least two the same digits.
The last databit of a frame can be a 0 or 1. However the sync pulse has to be exactly 18 bits long. Therefore, a preamble (Table 6) is put before the sync, which always ends on 00. The preamble also compensates the different number of bits in the odd and even frame pulse (Table Z), so that a full frame always takes 190 bits = 28.5 ms.

Table 6. Preamble for odd and even frame, ensuring a sync of exactly 18 1's.

frame	preamble
odd	1100
even	110000

Modulation

The final stage is the radio modulation. This step is rather straightforward. The preamble, sync pulse, odd/even code, and 160 6to10 coded frame bits are concatenated to form a **full frame** of 190 bits. Then, to remove DC in the radio signal (an unequal amount of 0's and 1's) all bits in the full frame are inverted regularly. Then the bits are modulated onto the frequency.

Remove DC

Removing DC is necessary in most radio transmitters, usually because of the use of voltage-controlled oscillators that drift towards the center frequency if provided a DC signal. Because the receiver doesn't drift, this would cause signal strength loss or worse. To avoid (well, mostly eliminate) DC, all channel bits (thus, including the preamble and sync) are inverted every third and fourth. So the bits in the first even and odd full frame are all straight up, the third and fourth full frame are all inverted, the fifth and sixth full frame are straight up again, etc.

Modulation

The bits are then modulated onto a radio wave. Generally, PCM uses Frequency Shift Keying or FSK. In FSK two frequencies close to the channel frequency are transmitted, one to transmit a 0 and the other to transmit a 1. Every bit takes 150 μs.

At the 72MHz band, Futaba uses the channel frequency f to transmit a 0, and f - 5kHz to transmit a 1 [Gulls03]. Channel distance is 20kHz on this band.
At the 35MHz band, channel spacing is only 10kHz and things are apparently set up somewhat different [Armitage03]. With a center frequency f, the transmitter transmits f-1500Hz for a 0, and f+1500Hz for a 1.
This modulation mechanism is according to literature, we did not measure it ourselves. For the reverse engineering of the PCM format we relied on the trainer output of the transmitters, which outputs the un-modulated full frames.

Acknowledgements

I would like to thank Angelos Gonias, Phil Cole, Erhard Klinke and Islander for their input and help with the reverse engineering and testing various transmitters.

Reference

[Armitage03] Armitage, D. (2003). 35 MHz Channel Spacing. Available Internet: http://rcrc.co.za/Other/35_mhz_channel_spacing.htm

[Autopilot02] autopilot: Do it yourself UAV. Available Internet: http://autopilot.sourceforge.net/index.html

[fmadirect03] FMA Direct (2003). FAQ's. FMA Direct, Frederick, MD. Available Internet: https://www.fmadirect.com/site/faqs.htm?category=1.

[Gulls03] Gulls, T. P. (2003). Everything You NEVER Wanted To Know About Radios! http://www.torreypinesgulls.org/Radios.htm

[Kassam99] Kassam, S. A. (1999). Cyclic Codes, and the CRC (Cyclic Redundancy Check) code. Part of course TCOM370 Notes 99-9, Principles of Data Communication, Dept,. of Electrical Engineering, Univ. Pennsylvania. Available Internet: http://www.seas.upenn.edu/~kassam

[Pasman03] PCM1024Z format: What's Known? Private Report. http://graphics.tudelft.nl/~wouter/publications/publ.html

[Pasman03b] Pasman, W. (2003). Latency of Futaba FF8s PPM and PCM Radio Controller. Private Report. http://graphics.tudelft.nl/~wouter/publications/publ.html

[Runryder03] PCM1024: Part II. http://www.runryder.com

[Sekiriki02] Sekiriki (2002). Re Does Futaba PCM1024 use CRC or checksum? Bulletin board answer on http://www.sekiriki.jp/smartpropo/index.html

[Sekiriki02b] SmartPropo (2002). Available Internet: www.sekiriki.jp.

Appendix A: Hard- and Soft-ware

This appendix describes the software and hardware that was used for reverse engineering.

Hardware

Our software (both macintosh and PC) uses a simple cable connecting the transmitter output to the computer audio inputs (Figure 6. Two resistors are needed to scale down the high transmitter signal voltage (typically above 8V) down to audio line signal level (1V).

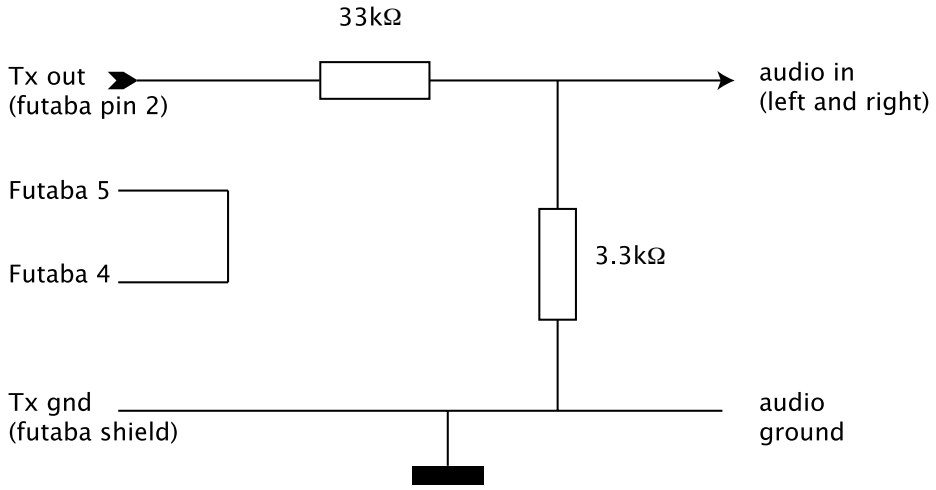

Figure 6. Wiring from transmitter to computer audio ports. Pin numbers are for Futaba transmitters. For Futaba, shortcut between pin 4 and 5 turns on the transmitter.

Software

The software we made is specifically designed to support the reverse engineering effords. It is not aimed at high efficiency, but instead assumes a powerful CPU (few hundred MHz) allowing vast amounts of printing and flexible general internal (but less optimized) code.

The basic sampling mechanism of both macintosh and PC versions of the software use hardware in the computer to sample the incoming signal at 44kHz. The signal is thresholded, and pulse lengths are calculated. These pulse lengths ar converted back into a string of 0's (signal below threshold) and 1's (signal above threshold).

After this point, the macintosh and PC version of the software differ. The Macintosh version was developed from scratch. There are four modules all compiled into a single program. The sampling module stores all bits into a data block until the next sync pulse comes by. After the sync pulse arrived, this module does a callback to the interpretation module, that interprets the bits as PCM data. After the interpretation module is done, that module calls the result handling module. The result handling module in our case will print the result, depending on the printing options selected. The last module responsible for interpreting the printing options and initializing all other modules.

Table 7 shows the options of the macintosh version. The program is started using a command from the terminal window "PCM [printoption]*". Options are separated by whitespace and start with either a '-' (minus) or '+' (plus). A minus turns an printoption off, a plus turns it on. After the plus or minus follows a letter to indicate the printoption targeted, and sometimes a few extra parameters. For example PCM -p0 turns off printing

of non-inverted frames. The a option needs some extra explanation. In general it looks like a^^^^: auxfields in the four pcm_packets are holding 4 digits. ^ can be 0,1,2 or 3. For example, PCM +p0 -a2020 prints all frames with polarity 0 and with NOT aux fields 2,0,2,0. You can set only 1 option for p, f for accept and one other for reject. So PCM +a0000 -a2000 is okay but PCM -a0000 -a1000 is not ok. Be careful to use the right digits, for instance digit 3 will be accepted for f but f=3 will never become 'true'. The option mechanism is rather clumsy and not as consistent as desirable, please refer to the source code for more details.

Table 7. Program printing options for Macintosh decoder software.

option letter	meaning	default printmode	extra params
p	polarity: whether bits in the full frame are inverted.	print all	0: non-inverted 1: inverted
f	odd or even frame	print both	0=even 1=odd
s	sync details	on	0=off 1=on
b	print all binary digits in frame	on	0=off 1=on
c	parsed packet info	on	0=off 1=on
t	binary aux fields	on	0=off 1=on
a	aux fields	not set	4 digits each 0,1,2 or 3.

The PC version uses separate sampling and interpretation programs. The sample program has the name ReadPCM. "ReadPCM T" tests if PCM signal is present. It will dump a line to the standardout every time a PCM frame comes by. "ReadPCM L FileName NbSamples" dumps frame data to a logfile. After the ReadPCM file has been stopped, the logfile still needs to be interpreted. The DecodePCM program just does that. The call is "DecodePCM test.log <verboselevel>" where <verboselevel> is 0, 1, 2, 3 or 4, higher values print more details.

B: Transmitter Specifics

We found that different transmitters in the Futaba line have different ways of generating those channels. This appendix lists a few peculiarities of the different transmitters

FF6

No zero-delta used.
No channel 7-10 available.

FF8 super

All failsafe frames in fixed order.
10 bit DACs
BFR bit goes high when throttle goes to below 27%
The non-realtime data is also transmitted immediately after a failsafe for one of the channels is turned to 'preset' but not when turned to 'normal'.
Channel 9 is turned on by pulling the snap/trainer switch.
throttle normal failsafe behaviour is throttle to idle
delta=0 is used.

FF 9Z-WC and 9Z

The BFS behaviour can be programmed, for instance to low or high throttle stick, or to switch A.
Channel 9: switch on gives 0, off gives 1.
Channel 10 is controlled by switch D. down gives 1, up gives 0.
No delta=0 used.
10 bit DACs

FC16

delta=0 not used. All positions are even.
9bit DACs.
Failsafe frames change order.
BFR bit is copied onto CH10
Throttle normal failsafe behaviour is throttle to center.

FC28

Channel 9: switch on gives 0, off gives 1.
Channel 10 is controlled by switch. off gives 1, on gives 0.

Appendix C: Techniques

Measuring the delta field is a bit tricky. To measure how the transmitter encodes specific jumps, it is necessary to provide those jumps at the input side of the transmitter. Applying those jumps in analog form into the potmeters as we did in our latency measurements [Pasman03b] might be prone to noise. Instead we came up with two techniques: the flip-switch and delayed-servo technique.

The Flip-switch Technique

The flip-switch technique mixes a stick position into a proportional channel. The absolute position jump caused by switching the switch can be accurately determined by reading out the channel with our software, and the mixing can be adjusted until the required jump is caused. Then, we flip the switch until we find a delta code corresponding to that switch jump. This gives us the delta code associated with the absolute jump we set up.

The Delayed-servo Technique

The delayed-servo technique uses the servo delay function available in the FF8 and FF9 transmitters. Similar to the flip-switch technique we set up a jump associated to a switch, mixing into a proportional channel. Now we set up for as large a jump as possible. Next, we set up delayed servo jumps. This is a feature that smoothens out large jumps by interpolating inbetween values (this is especially useful for instance for having a landing gear coming down slowly). It shows that this feature uses equal steps between two frames. The actual step size is determined with our software. Because there is a delta code between every two position codes for the channel being affected, we have to divide the difference between the two position codes by two in order to find the jump between the position code and the delta code. Care has to be taken that the difference between the two position codes is even, to ensure both jumps are equal size. This technique is

especially useful when checking the delta behaviour over failsafe frames. The technique is less suited for very small and very large jumps, because of limitations on the delay.

Appendix D: Decoder Diagram

Figure 7 shows how the decoding can be understood as moving incoming data packets to the appropriate memory position representing the channel or failsafe position. It is convenient to see that the decoding is straightforward in this scheme, it gives some evidence that we have all the bit signs right.

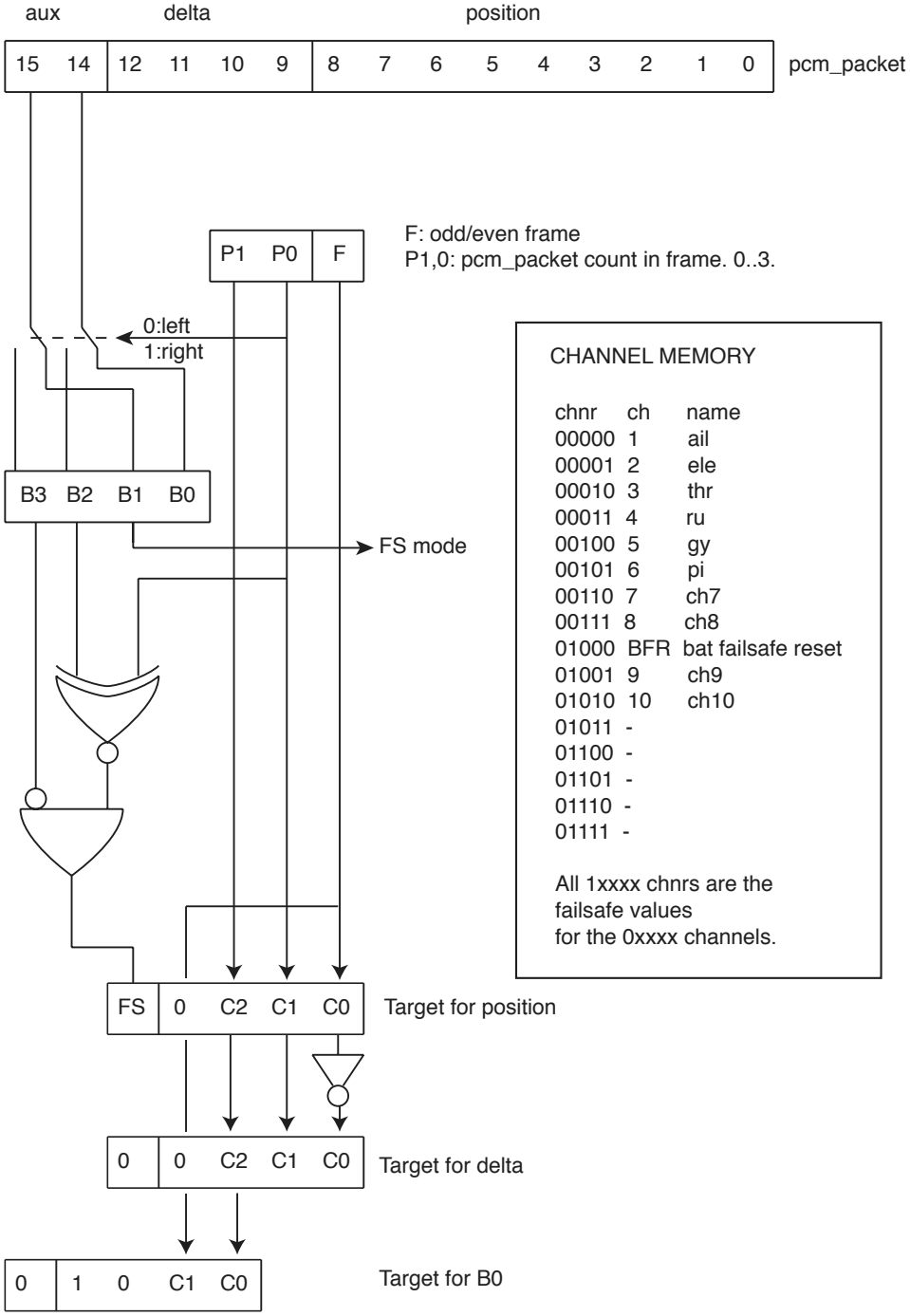

Figure 7. Diagram showing PCM decoding as a data-to-memory strategy.

Supplement

10 february 2008

I received a report from Olav that Table 3 (the XOR table for calculating the CRC) might be in reverse order. For me Table 3 worked (and for several others as well), but Olav had to reverse the order, so A1 => 4a, A0 => 25 and so on. Olav also provided the VHDL code for checksum computation, as shown in Figure 1, includng a few lines of explanation:

> "Temp holds the frame consisting of aux, delta pos and absolute pos. The for-loop scans through temp and xor ecc if temp(i) is one. The forloop initiates I at 15 and counts downto 0. So: temp(15) is auxbit 1, and the corresponding ecc_table-value (located at ecc_table(15) is 4A. Olav I might have mixed up the "to" and "downto" here, but the code shown is tested and does work."

```
type ECC_TABLE_TYPE is array (15 downto 0) of std_logic_vector(7
downto 0);
constant ECC_TABLE : ECC_TABLE_TYPE :=
(X"4A",X"25",X"A7",X"E6",X"73",X"8C",X"46",X"23",

X"A4",X"52",X"29",X"A1",X"E5",X"C7",X"D6",X"6B");

function calc_ecc(constant aux:   in std_logic_vector(1 downto 0);
                  constant delta: in std_logic_vector(3 downto 0);
                  constant pos:   in std_logic_vector(9 downto 0))
return
std_logic_vector is

  variable temp: std_logic_vector(15 downto 0);
  variable ecc : std_logic_vector( 7 downto 0);
  begin
    temp := aux & delta & pos;
    ecc  := (others => '0');

    for i in 15 downto 0 loop
      if temp(i) = '1' then
        ecc := ecc xor ECC_TABLE(i);
      end if;
    end loop;
    return ecc;
  end function calc_ecc;
```

Figure 1. VHDL code for the checksum calculation, provided by Olav.