What is the iTelemetry protocol?

Back in 2008 ImmersionRC was heavily focussed on long range flight with fixed-wing aircraft. As part of this focus, we developed a telemetry protocol which could be transmitted on the audio channel of any A/V link, and could be used to provide GPS coordinates for antenna tracking, and model recovery, plus transmit telemetry for battery level, current draw, etc. This old vimeo video explains a bit about what we could do back then: https://vimeo.com/2324907

This protocol was implemented in the EzOSD, our EzAntennaTracker, and the telemetry applications for iOS and Android devices.

It was simple enough that it could be decoded by either a hardware or software decoder, and in the case of the smartphone applications it could be fed into the mic port of most devices, and treated by the audio input systems of these devices.

Now fast forward a decade, and we still have many customers using this protocol, who wish to embed it into their own systems. We also have requests from developers who wish to evolve our aging smartphone apps, and fix issues created by the evolution of operating systems in these devices.

So, we have decided to open up the protocol for non-commercial use.

The Details…

The protocol is a simple manchester-encoded serial stream at 4777 baud.
Yes… 4777 instead of the more 'typical' 4800 baud. Some history behind this decision that is not relevant today.

Each packet consists of the following structure

Preamble	8 bytes of 0xAA
Sync	2 bytes of 0xCC
header	2 bytes
Data	max. 40 bytes
Crc	1 byte
Checksum	1 byte

Preamble 8 bytes 0xaa	Sync 2 bytes 0xcccc	Header 2 bytes	Data Max 40 bytes	Crc 1 byte	Checksum 1 byte

Between packets a continuous stream of bit toggling ensures that the average DC level of the telemetry waveform remains the same outside of a packet as inside.

ezTelemetry.h

```
//---------------------------------------------------------------------------------------------------------------
//
// Project: EzOSD, telemetry module
//
// Component: Use the high bandwidth of Airwave Rx/Tx modules to transmit telemetry down one of the
//                                two stereo audio channels.
//
// Started: Oct 2008
//
// NOTE: This code is formatted with 4 character tab spacing, not 3. This may be
// changed in most editors (for CCS it's under options->editor properties)
//
// TODO: Manchester encoding would be nice
//                    For this, a '1' is encoded as a 1 for the 1st 1/2 of the bit cell, and a '0' for the second 1/2
//                    a '0' is encoded as as 0 for the 1st 1/2 of the bit cell, and a '1' for the second 1/2
//
//---------------------------------------------------------------------------------------------------------------

#include "..\shared\fastcrc.h"

//#define SECOND_TELEMETRY_PIN

typedef enum { TELEMETRY_Slow,
                                TELEMETRY_Normal,
                                TELEMETRY_Double } TELEMETRYRATE;

// definition of the packet which will be sent over the wire
// NOTE: Some loops are hand-coded, if these are changed, check the code below
#define PREAMBLE_BYTES                      8          // preamble includes bytes with alternating 1's and 0's, used to sync up the Rx to the Tx
#define SYNC_BYTES                          2          // synchronization bytes, pairs of 1's and 0's, used to sync the de-serializer to byte boundaries
#define HDR_BYTES                           2          // storage for packet len, and packet ID
#define HDR_INDEX                           (PREAMBLE_BYTES + SYNC_BYTES)
#define CRC_BYTES                           1          // the CRC, just one byte
#ifdef EXTRA_TELEMETRY_CHECKSUM
#define CSUM_BYTES                          1          // the checksum, just one byte
#else
#define CSUM_BYTES                          0          // the checksum, just one byte
#endif
#define DATA_INDEX                          (PREAMBLE_BYTES + SYNC_BYTES + HDR_BYTES)
#define FIXED_PACKET_LEN        (PREAMBLE_BYTES + SYNC_BYTES + HDR_BYTES + CRC_BYTES + CSUM_BYTES)          // length of the fixed part of the packet

#define MAX_PACKET_LEN                      35                      // needs to be large enough for the largest packet that we'll send out
int8 rfPacket[MAX_PACKET_LEN];                  // packet definition, used for Rx and Tx
int totalPacketLen = 0;

#define PREAMBLE_BYTE           0b10101010
#define SYNC_BYTE                           0b11001100
#define SYNC_WORD                           0b1100110011001100

int baudRateTimerVal = 156;
int baudRatePostScale = 1;
int16 telemetryPin = OSD_TELEMETRY;
#ifdef SECOND_TELEMETRY_PIN
int16 telemetryPin2 = OSD_TELEMETRY_TEL1;
#endif

#define PACKET_ID_GPS                                   1
#define PACKET_ID_BATTERY                               2
#define PACKET_ID_LINKSTATUS                3

//---------------------------------------------------------------------------------------------------------------
void telemetrySetRate(TELEMETRYRATE rate)
{
           switch(rate)
           {
                      case TELEMETRY_Slow:
#ifdef SYSTEM_CLOCK_64MHZ
                                baudRateTimerVal = 207;                     // approx. 2400 baud    (416us measured in the simulator)
#else
                                baudRateTimerVal = 156;                     // approx. 2400 baud
#endif
                                baudRatePostScale = 2;
                                break;

                      case TELEMETRY_Normal:
#ifdef SYSTEM_CLOCK_64MHZ
                                baudRateTimerVal = 206;                     // approx. 4800 baud    (207.875us measured in the simulator)
#else
                                baudRateTimerVal = 156;                     // approx. 4800 baud
#endif
                                baudRatePostScale = 1;
                                break;
```

```
                    case TELEMETRY_Double:                                  // approx. 9600 baud
#ifdef SYSTEM_CLOCK_64MHZ
                                baudRateTimerVal = 103;     // (103.93us measured in the simulator)
#else
                                baudRateTimerVal = 78;
#endif
                                baudRatePostScale = 1;
                                break;
          }

          // setup the baud rate generator timer, timer1 is a 16-bit timer
          // To generate a 9600 baud rate
          // 1/9600 = 104.166666 us / 0.66us = 157.8282, call it 158
                    setup_timer_2 (T2_DIV_BY_16, baudRateTimerVal, baudRatePostScale);                      // 48MHz / 4 = 12MHz / 16 = 1.3us
}

//-------------------------------------------------------------------------------------------------------------------------
void telemetryInit(bool bEnable)
{
          // set the telemetry output line to an appropriate state
                    if(bEnable)
        {
                                output_high(telemetryPin);
            #ifdef SECOND_TELEMETRY_PIN
            output_high(telemetryPin2);
            #endif
        }
                    else
        {
                                output_float(telemetryPin);                 // leave the audio line alone
            #ifdef SECOND_TELEMETRY_PIN
            output_float(telemetryPin2);
            #endif
        }

                    // looks like a kludge, but uses less code than the loop above!
                    rfPacket[0] = PREAMBLE_BYTE;
                    rfPacket[1] = PREAMBLE_BYTE;
                    rfPacket[2] = PREAMBLE_BYTE;
                    rfPacket[3] = PREAMBLE_BYTE;
                    rfPacket[4] = PREAMBLE_BYTE;
                    rfPacket[5] = PREAMBLE_BYTE;
                    rfPacket[6] = PREAMBLE_BYTE;
                    rfPacket[7] = PREAMBLE_BYTE;

                    rfPacket[8] = SYNC_BYTE;
                    rfPacket[9] = SYNC_BYTE;
}

int iPacketIndex = 0;
int sendingByte = 0;
int iBitCount = 8;
int1 iManchState = 0;
int1 bLastBitSent = 1;                              // must init. to 1, not zero
int1 bInInterPacketIdle = 1;               // TODO: check default state
int iInterPacketBitState = 0;

//-------------------------------------------------------------------------------------------------------------------------
#INT_TIMER2 HIGH                // Needs to be a fast interupt (called at 15kHz or so...)
void baudRateGeneratorInterrupt()
{
          if(bInInterPacketIdle)
          {
                    // we are between packets, keep toggling the data line at a rate of 1/2 the highest rate
                    if(iInterPacketBitState & 0x02)
                    {
                              output_low(telemetryPin);
                              #ifdef SECOND_TELEMETRY_PIN
                              output_low(telemetryPin2);
                              #endif
                    }
                    else
                    {
                              output_high(telemetryPin);
                              #ifdef SECOND_TELEMETRY_PIN
                              output_high(telemetryPin2);
                              #endif
                    }

                    ++iInterPacketBitState;
          }
          else if(bLastBitSent)
          {
                    output_high(telemetryPin);
        #ifdef SECOND_TELEMETRY_PIN
                    output_high(telemetryPin2);
              #endif

                    bInInterPacketIdle = 1;
                    iInterPacketBitState = 0;

                    // Nov 2012: Don't disable interrupts here, we are going to keep sending lower freq. telemetry to keep the AGCs/Rx happy
                    //disable_interrupts(INT_TIMER2);
```

```
		}
		else
		{
			// simply clock out each bit of the packet, serially, MSB first
			// when done, disable interrupts
				if(iManchState == 0)
				{
					if(sendingByte & 0x80)
		{
						output_low(telemetryPin);
			#ifdef SECOND_TELEMETRY_PIN
				output_low(telemetryPin2);
			#endif
		}
					else
		{
						output_high(telemetryPin);
			#ifdef SECOND_TELEMETRY_PIN
				output_high(telemetryPin2);
			#endif
		}

					++iManchState;
				}
				else
				{
					if(sendingByte & 0x80)
		{
						output_high(telemetryPin);
			#ifdef SECOND_TELEMETRY_PIN
				output_high(telemetryPin2);
			#endif
		}
					else
		{
						output_low(telemetryPin);
			#ifdef SECOND_TELEMETRY_PIN
				output_low(telemetryPin2);
			#endif
		}
					sendingByte <<= 1;

					// did we reach the end of a byte?
						// enable manchester encoding only when we're past the preamble and sync
						if(--iBitCount == 0)
						{
							iBitCount = 8;
							if(++iPacketIndex == totalPacketLen)
								bLastBitSent = 1;
							else
								sendingByte = rfPacket[iPacketIndex];		// get the next byte
						}
						if(iPacketIndex >= HDR_INDEX)
							iManchState = 0;
				}
		}
}

//-------------------------------------------------------------------------------------------------------------------
// note:
//	nBytes: # bytes in the data, excluding space required by the packet ID, and length bytes
void telemetrySend(int nBytes, int packetId, void *pData)
{
		// ensure that we don't collide
			if(!bLastBitSent)
				return;		// just jetisson the
packet, nobody will miss it

		// transfer payload into rfPacket
			rfPacket[HDR_INDEX] = packetId;
			rfPacket[HDR_INDEX+1] = nBytes;				// packet len, excluding header length

			memcpy(&rfPacket[DATA_INDEX], pData, nBytes);

		// CRC, which includes the packetId, and number of bytes
			rfPacket[DATA_INDEX + nBytes] = crcFast(&rfPacket[HDR_INDEX], nBytes + HDR_BYTES);

#ifdef EXTRA_TELEMETRY_CHECKSUM
		// Checksum, in addition to the CRC, double protection for better noise immunity
			rfPacket[DATA_INDEX + nBytes + 1] = checksumFast(&rfPacket[HDR_INDEX], nBytes + HDR_BYTES);
#endif

		// initialize some stuff
			iBitCount = 8;
			iPacketIndex = 0;
			sendingByte = rfPacket[iPacketIndex];
			totalPacketLen = FIXED_PACKET_LEN + nBytes; // total len = fixed + variable
			iManchState = 1;		// init to high, we stay here until we've clocked
out syncs + preamble
			bLastBitSent = 0;
			bInInterPacketIdle = 0;

		// kick off the timer
```

```
                    set_timer2(baudRateTimerVal);                                   // schedule the callback
                    enable_interrupts(INT_TIMER2);                                  // (turned off automatically once sent)
}
```

CRC Calculator:

```
//---------------------------------------------------------------------------------------------------------------------
// returns a 8-bit CRC for the supplied buffer
uint8_t crcFast(uint8_t* buf, uint8_t size)
{
    uint8_t i, j;
    uint8_t crc = 0;

    for (i = 0; i < size; i++)
    {
        crc = crc ^ buf[i];

        for (j = 0; j < 8; j++)
        {
            if ((crc & 0x80) != 0)
            {
                crc <<= 1;
                crc ^= 0x07;
            }
            else
            {
                crc <<= 1;
            }
        }
    }
    return (crc);
}
```