ACLZ log format

Altimeter Cloud

Software, filters & technical · v1.0

https://www.altimetercloud.com/support/software-and-technical/aclz-log-format/

Your flight data: the .aclz file

On altimeters that support it, every flight is saved as a single .aclz file. One file holds the whole flight: every sample from every sensor, the events the device detected, and a record of how it was configured when it flew. Nothing is stored anywhere else and nothing is left behind on the device, so the file you copy off is the flight, complete.

It is also signed. The device signs the contents before compressing them, so a single altered byte anywhere in the file is detectable. That is what makes an .aclz worth submitting as evidence for a record or a certification attempt: not just your data, but your data provably as flown.

The Nano and the Jupiter both write this format. They share the container, the compression, the signing scheme and the way values are packed. What differs is the shape of the data inside, because a Nano records one table at one rate and a Jupiter records six tables at six rates. This page covers the parts that are the same on both, then the parts that are not.

Three ways to get at it

Upload it to altimetercloud.com

Charts, analysis, comparison against your other flights, and CSV export in real units. Nothing to install. This is the route almost everyone wants.

The desktop converter, coming soon

A Windows application that opens an .aclz, shows you the flight, and writes CSV. It works offline, checks both signatures, and handles Nano and Jupiter files. In testing now, released in the coming months.

Read it yourself

The format is documented below and the reference converter is a single Python file with no dependencies. For anyone building their own tooling, or who simply wants to know what is in there.

What every device does the same way

An .aclz is one LZMA compressed stream, the 7-Zip algorithm family, which standard tools read directly. Decompress it and you get a fixed-size header followed by the flight data stored in columns.

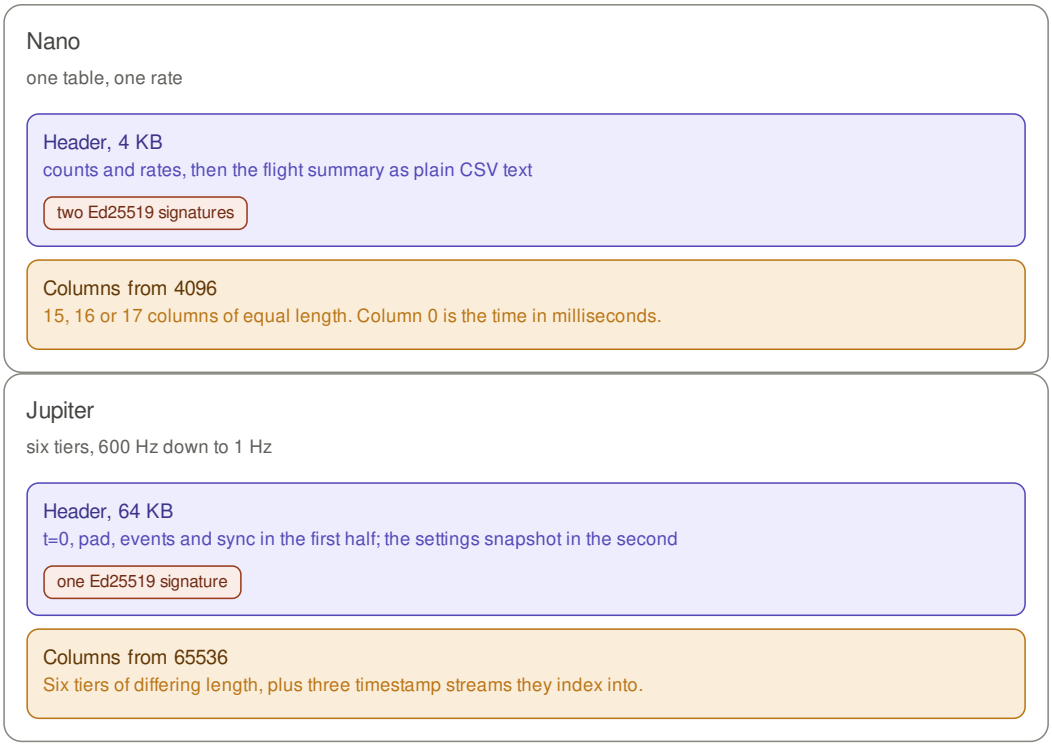
Storing by column means one field's entire time series sits together, all the altitudes, then all the temperatures, rather than sample after sample of mixed values. Similar numbers end up next to each other, which is exactly what a compressor exploits. Each column then stores the difference between consecutive samples rather than the values themselves, so a reading that barely changes costs a single byte. At regular intervals an absolute value is written instead of a difference; these keyframes bound how far an error can travel, because a pure difference chain would carry a bad value forward to the end of the column.

Values are packed as varints: seven bits per byte, low bits first, the top bit meaning another byte follows. Signed numbers are zigzag encoded first so that small negatives stay small, which makes the decode (u >> 1) ^ -(u & 1). A missing or not-yet-known value is stored as INT32_MIN.

The signature works the same way everywhere. Before compressing, the device takes a SHA-256 digest of the complete uncompressed contents with the signature slot zeroed, signs that digest with Ed25519, and writes the signature into its slot. To verify, decompress, zero the same slot, hash, and check. Any edit anywhere fails.

What differs between devices

Two things: the size of the header and what fills it, and whether the data is one table or several. Everything else above is shared.



Same container, same encoding, same signing. Different shape inside.

On a Nano, time is simply the first column, in milliseconds, and every column is the same length, so there is nothing to index. The header carries the sample count, the sample rate, the column count and the keyframe interval as consecutive 32-bit integers, which means the column block of any Nano file can be walked without knowing the version first, even though what each column *means* still depends on it. The rest of the header is the flight summary as plain CSV text: apogee, times, velocities, serial number, firmware, and the settings string as flown. Nano files also carry a second signature, over the CSV the log expands to, so an exported CSV can be checked on its own without the original .aclz.

On a Jupiter, each tier has its own sample count and its own keyframe interval, and time comes from three timestamp streams rather than a column: the 600, 300 and 100 Hz tiers share the first, 50 Hz has its own, and 10 and 1 Hz share the third. A slower tier indexes into its group's stream at a fixed stride, so a 300 Hz sample j uses entry 2j, a 100 Hz sample j entry 6j, and a 1 Hz sample j entry 10j of the third. All three streams tick the same 10 microsecond clock, so any value still lays over any other exactly.

Header offsets

Everything a decoder needs sits at a fixed offset in the decompressed file. Check the version word before touching the columns: column counts and value scales change between versions, and the data cannot tell you itself.

Offset	Nano	Jupiter
0	ASCII magic string, identical on both	
32	format version, 32-bit	
36	product code, 32-bit	t=0, 32-bit tick count
40	sample count	pad elevation, float
44	sample rate	six per-tier sample counts, 32-bit
48	column count, 15/16/17	
56	keyframe interval	
68		six per-tier keyframe intervals, 16-bit
128	Ed25519 signature over the file, 64 bytes	
192	Ed25519 signature over the CSV	flight event block

256	flight summary, CSV text, NUL padded	
32768		SET1, length, then key=value pairs split on 0x1F
columns	from 4096	from 65536

Columns are pure sequences of varints, one column completely before the next, in a fixed order. On a Nano that is simply column 0 through to the last. On a Jupiter it is the first timestamp stream, then every 600 Hz field, every 300 Hz field, every 100 Hz field, the second timestamp stream, the 50 Hz fields, the third timestamp stream, the 10 Hz fields, and the 1 Hz fields. Reassembly is the keyframe rule: within a column, sample i is an absolute value when i % kf == 0 for that table's keyframe interval, and otherwise a difference to add to the previous decoded value. For Jupiter format version 9 the field counts per tier are 3, 7, 7, 8, 18 and 49, not counting the timestamp streams.

Two details worth getting right. Both products share the magic string, so a converter accepting either has to look further: the Nano writes a product code at offset 36 and always has 15, 16 or 17 columns, while the Jupiter has t=0 at that offset and no product code, so recognise the Nano positively and treat anything else as a Jupiter. And Jupiter timestamps are unsigned 32-bit ticks stored in signed columns, so subtract t=0 with a 32-bit wrap rather than plainly, or a device powered for more than about six hours produces times wrong by billions. Samples from before launch decode as negative, which is correct: the pre-roll buffer was recorded before t=0.

The reference converter

One Python file, no dependencies beyond the standard library, and the whole workflow in one command:

python3 aclz_to_csv.py flight_1234.aclz

It works out which device wrote the file and handles both. A Jupiter gives you six tier CSVs and the settings snapshot as settings.txt; a Nano gives you samples.csv and the flight summary as metadata.csv. The header and settings decode for any version, the Jupiter column set is written for version 9, and any Nano column count is read from the header.

It deliberately writes the raw quantised integers the device stored rather than converting to real units, because which column means what, and what scale it carries, is version territory. If you want a CSV in metres and m/s, use altimetercloud.com or the desktop application. If you are building tooling, raw integers are what you want anyway.

This is the only complete implementation we publish. Every full script in another language is another parser to keep in sync with a format that is still growing, so port from this one when you need another stack, and check the version word first in anything you build.

This code is Python: aclz_to_csv.py, the complete converter

```
#!/usr/bin/env python3
# aclz_to_csv.py - reference converter for AltimeterCloud .aclz flight logs.
#
# Handles both file shapes:
#   Jupiter   64 KB header, six data tiers at 600/300/100/50/10/1 Hz, three
#             timestamp streams. Writes tier600.csv .. tier1.csv + settings.txt.
#   Nano      4 KB header, one table at one rate, time carried as a column.
#             Writes samples.csv + metadata.csv.
#
# Values are the raw quantised integers the device stored. Field meanings and
# scales are defined per format version, so they are deliberately not applied
# here; use altimetercloud.com or the desktop converter for CSVs in real units.
# The time column is seconds from t=0 for Jupiter, milliseconds since power-on
# for Nano, exactly as each device stores it.
#
#     python3 aclz_to_csv.py flight_1234.aclz
import lzma, struct, sys

INT32_MIN = -2147483648


def read_varint(buf, pos):
    u = 0; s = 0
```

```
    while True:
        b = buf[pos]; pos += 1
        u |= (b & 0x7F) << s
        if not (b & 0x80):
            return u, pos
        s += 7


def unzig(u):
    return (u >> 1) ^ -(u & 1)


def decode_column(buf, pos, n, kf):
    col = []; prev = 0
    for i in range(n):
        u, pos = read_varint(buf, pos)
        v = unzig(u)
        x = v if (kf == 0 or i % kf == 0) else prev + v
        col.append(x); prev = x
    return col, pos


def write_csv(name, first_name, first_col, cols):
    with open(name, 'w') as f:
        f.write(first_name + ',' + ','.join('f%d' % i for i in range(len(cols))) + '\n')
        for r in range(len(first_col)):
            row = ['' if c[r] == INT32_MIN else str(c[r]) for c in cols]
            f.write('%s,%s\n' % (first_col[r], ','.join(row)))


def identify(body):
    """
    Which product wrote this file.

    Both products share the magic string, so the words after it decide. A Nano
    puts its product code at offset 36 and always writes 15, 16 or 17 columns,
    recorded at offset 48. A Jupiter puts t=0 at offset 36 and has no product
    code, so the test is written to recognise the Nano positively and treat
    everything else as a Jupiter.
    """
    ver = struct.unpack_from('<I', body, 32)[0]
    second = struct.unpack_from('<I', body, 36)[0]
    nfields = struct.unpack_from('<I', body, 48)[0]
    if ver in (1, 2) and second == 1 and nfields in (15, 16, 17):
        return 'nano'
    return 'jupiter'


def convert_nano(body):
    ver, product, ns, rate, nf, comp, kf = struct.unpack_from('<7I', body, 32)
    print('Nano  format v%d  %d samples at %d Hz  %d columns' % (ver, ns, rate, nf))

    # The 4 KB header carries the flight summary as plain CSV text, NUL padded:
    # apogee, times, velocities, serial number, firmware, settings string and so
    # on. It is the same block the device writes at the top of a CSV export.
    text = body[256:4096].split(b'\x00')[0].decode('utf-8', 'replace')
    with open('metadata.csv', 'w') as f:
        f.write(text)
```

```
    print('metadata.csv: %d bytes of flight summary' % len(text))

    # One table, every column the same length. Column 0 is the timestamp in
    # milliseconds, so unlike the Jupiter there is no separate stamp stream.
    pos = 4096
    cols = []
    for _ in range(nf):
        c, pos = decode_column(body, pos, ns, kf)
        cols.append(c)

    write_csv('samples.csv', 't_ms', cols[0], cols[1:])
    print('samples.csv: %d rows; %d bytes of column data consumed' % (ns, pos - 4096))


def convert_jupiter(body):
    ver, t0 = struct.unpack_from('<II', body, 32)
    (floor_m,) = struct.unpack_from('<f', body, 40)
    counts = struct.unpack_from('<6I', body, 44)      # n600 n300 n100 n50 n10 n1
    kfs = struct.unpack_from('<6H', body, 68)         # keyframe interval per tier
    print('Jupiter  format v%d  t0 tick=%d  pad=%.1f m  samples=%s'
          % (ver, t0, floor_m, list(counts)))

    if body[32768:32772] == b'SET1':
        (slen,) = struct.unpack_from('<I', body, 32772)
        pairs = body[32776:32776 + slen].decode('utf-8', 'replace').split('\x1f')
        with open('settings.txt', 'w') as f:
            f.write('\n'.join(p for p in pairs if p) + '\n')
        print('settings.txt: %d entries' % sum(1 for p in pairs if p))

    if ver != 9:
        sys.exit('column decode here is written for format v9; this file is v%d - '
                 'update the column table before use' % ver)

    n600, n300, n100, n50, n10, n1 = counts
    pos = 65536

    sA, pos = decode_column(body, pos, n600, kfs[0])   # stamps: 600/300/100 group
    c600 = []
    for _ in range(3):
        c, pos = decode_column(body, pos, n600, kfs[0]); c600.append(c)
    c300 = []
    for _ in range(7):
        c, pos = decode_column(body, pos, n300, kfs[1]); c300.append(c)
    c100 = []
    for _ in range(7):
        c, pos = decode_column(body, pos, n100, kfs[2]); c100.append(c)
    sB, pos = decode_column(body, pos, n50, kfs[3])    # stamps: 50 group
    c50 = []
    for _ in range(8):
        c, pos = decode_column(body, pos, n50, kfs[3]); c50.append(c)
    sC, pos = decode_column(body, pos, n10, kfs[4])    # stamps: 10/1 group
    c10 = []
    for _ in range(18):
        c, pos = decode_column(body, pos, n10, kfs[4]); c10.append(c)
    c1 = []
    for _ in range(49):
        c, pos = decode_column(body, pos, n1, kfs[5]); c1.append(c)
```

```
    # Stamps are unsigned 32-bit ticks stored as signed, so subtract with a
    # wrap: after about six hours of uptime a plain subtraction goes negative
    # by billions. Pre-launch samples legitimately come out negative, because
    # the pre-roll happened before t=0.
    def secs(stamps):
        out = []
        for s in stamps:
            d = (s - t0) & 0xFFFFFFFF
            if d >= 0x80000000:
                d -= 0x100000000
            out.append('%.5f' % (d / 100000.0))
        return out

    write_csv('tier600.csv', 't_s', secs(sA), c600)
    write_csv('tier300.csv', 't_s', secs([sA[2 * j] for j in range(n300)]), c300)
    write_csv('tier100.csv', 't_s', secs([sA[6 * j] for j in range(n100)]), c100)
    write_csv('tier50.csv',  't_s', secs(sB), c50)
    write_csv('tier10.csv',  't_s', secs(sC), c10)
    write_csv('tier1.csv',   't_s', secs([sC[10 * j] for j in range(n1)]), c1)
    print('six tier CSVs written; %d bytes of column data consumed' % (pos - 65536))


def main(path):
    with open(path, 'rb') as f:
        body = lzma.decompress(f.read(), format=lzma.FORMAT_ALONE)

    magic = body[0:24].split(b'\x00')[0].decode('utf-8', 'replace')
    if not magic.startswith('ALTIMETERCLOUD-ACL'):
        sys.exit('not an ACL file (magic: %r)' % magic)

    if identify(body) == 'nano':
        convert_nano(body)
    else:
        convert_jupiter(body)


if __name__ == '__main__':
    main(sys.argv[1] if len(sys.argv) > 1 else 'flight.aclz')
```

Porting it

The decompress entry point plus three primitives, varint reading, zigzag decoding, and delta-with-keyframe reassembly, are everything the format needs. They are identical on every device that writes this format; only the header offsets and the column table change. In Perl, noting the zigzag: use the branch form, since bitwise XOR is unsigned in some languages, Perl included.

This code is Perl: the decompress entry point and the port primitives

```
# Entry point: slurp the .aclz and decompress. Compress::Raw::Lzma reads the
# LZMA-alone stream directly (or shell out to: xz --decompress --format=lzma).
use strict; use warnings;
use Compress::Raw::Lzma;
sub read_aclz {
  my ($path) = @_;
  open(my $fh, '<:raw', $path) or die "open $path: $!";
  my $comp = do { local $/; <$fh> };
  close $fh;
  my ($lz, $st) = Compress::Raw::Lzma::AloneDecoder->new(AppendOutput => 1);
  my $body = '';
  $lz->code($comp, $body);
```

```
  return \$body;            # ref to the decompressed header + columns
}

# The three primitives every port needs. Identical for Jupiter and Nano.
sub read_varint {           # ($bufref, $pos) -> ($value, $newpos)
  my ($buf, $pos) = @_; my ($u, $s) = (0, 0);
  while (1) {
    my $b = ord(substr($$buf, $pos++, 1));
    $u |= ($b & 0x7F) << $s;
    return ($u, $pos) unless $b & 0x80;
    $s += 7;
  }
}
sub unzig {                 # zigzag decode; branch form, not XOR: Perl's ^ is unsigned
  my $u = shift; return ($u & 1) ? -(($u + 1) >> 1) : ($u >> 1);
}
sub decode_column {         # ($bufref, $pos, $n, $kf) -> (\@col, $newpos)
  my ($buf, $pos, $n, $kf) = @_; my @col; my $prev = 0;
  for my $i (0 .. $n - 1) {
    (my $u, $pos) = read_varint($buf, $pos);
    my $v = unzig($u);
    my $x = ($kf == 0 || $i % $kf == 0) ? $v : $prev + $v;
    push @col, $x; $prev = $x;
  }
  return (\@col, $pos);
}
```

What changes, and what does not

The container, the compression, the signing scheme, the varint and zigzag packing, and the delta-with-keyframe rule are stable. Anything written against those will keep working.

What moves is the column table: which fields a device records, in what order, and at what scale. New sensors and new features add columns, and that is what the format version word exists to tell you. A decoder that reads the version and refuses politely when it meets a number it was not written for is doing the right thing; one that assumes and carries on produces numbers that look plausible and are wrong, which is worse than an error message.

Files you already have keep decoding. A newer device writing a newer version does not change what an older file contains.