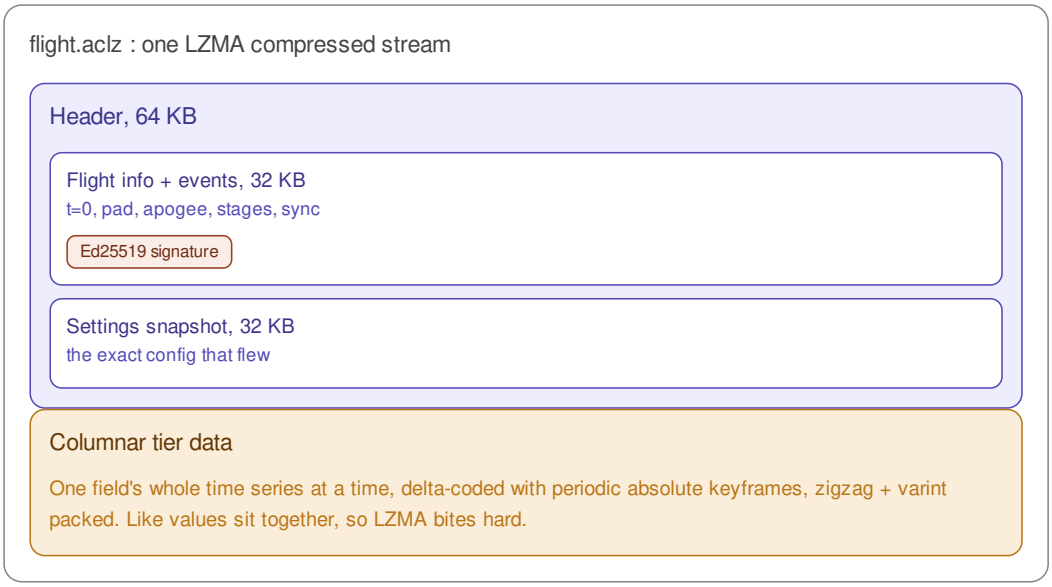
ACLZ log format

The ACL flight log format

Every flight Jupiter records becomes a single .aclz file. It is completely self-contained: the sensor data across every rate, the detected flight events, the sensor synchronisation results, and a snapshot of the exact settings the device flew with are all inside one file, cryptographically signed so any alteration is detectable. A full-rate flight costs roughly 100 KB per minute after compression, which is what makes uploading whole flights over a cellular link practical.

The same container, the same compression and the same signing scheme are used across the range. What differs between devices is the shape of the data inside: how many tables there are, at what rates, and which columns each holds. This page describes the Jupiter in detail because it is the richest example, and the Other devices section below covers how the Nano differs. A converter written against one is most of the way to reading the other.



Signed before compression: decompress, hash, verify. Any edited byte fails.

The container

An .aclz file is one LZMA compressed stream (the 7-Zip algorithm family, readable by standard tools) wrapping two things: a fixed header, and the flight data itself stored in columns. On the Jupiter that header is 64 KB. The file is signed before it is compressed: the device computes a SHA-256 digest of the complete uncompressed contents with the signature slot zeroed, signs the digest with Ed25519, places the signature into its slot in the header, and only then compresses. Verification runs the same steps in reverse, so a single altered byte anywhere in the file, a value, a timestamp, an event, fails the check.

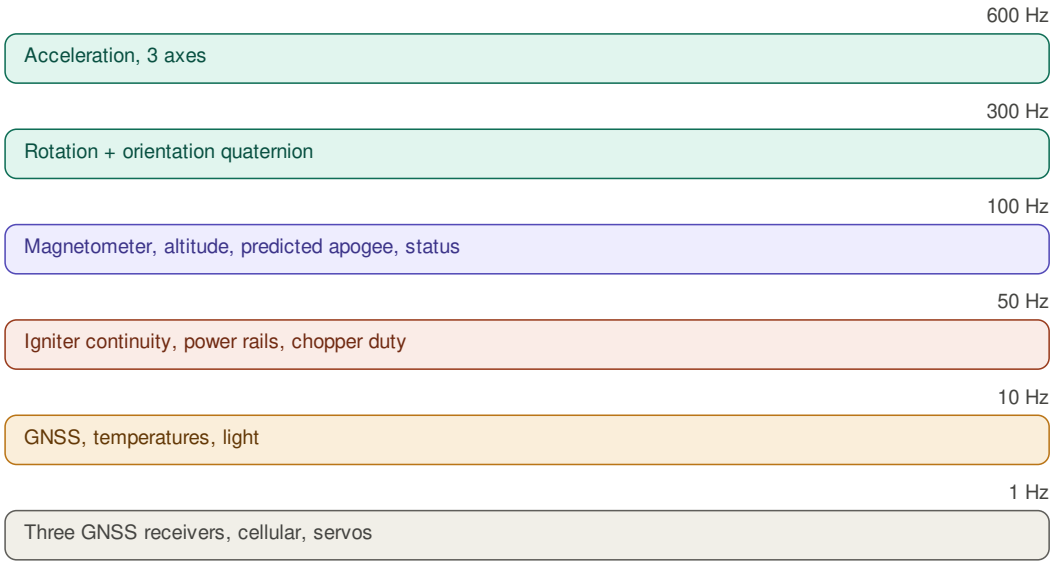
The header

The first half of the header holds the flight itself: the true launch time t=0, the pad elevation, per-tier sample counts, the landing result, every detected burnout and stage-ignition timestamp, the apogee record, the barometer-to-accelerometer synchronisation measured on that flight, and the Ed25519 signature. Space is also reserved for the flight event log, a timeline of outputs, protections and faults arriving in a future firmware. The second half is a complete snapshot of the device's configuration at the moment of saving, every setting and rule as flown, which means a flight can always be interpreted against the exact configuration that produced it, even years and many settings changes later.

The tier system

Jupiter records six data tiers at rates matched to how fast each quantity actually changes, from 600 Hz at the top to 1 Hz for slow housekeeping. The exact fields in each tier are still growing as features land, and the field lists are defined per format version rather than promised here, but the current shape is: acceleration at 600 Hz; rotation and orientation at 300 Hz; magnetometer, altitude and predicted apogee at 100 Hz; igniter continuity and power at 50 Hz; GNSS, temperatures and light at

10 Hz; and deep GNSS, cellular and servo detail at 1 Hz. Altitude at 100 Hz is stored as the raw barometric series alongside a TruePath correction term, so both the measurement and the filtered result are recoverable from the same file. Every sample is timestamped from one master clock with 10 microsecond resolution, and the timestamps travel as three streams rather than one: the 600/300/100 Hz tiers share the first, 50 Hz carries its own, and 10/1 Hz share the third. Three exist because the sample-ratio system can run the tier groups at different effective rates, so each group owns its time axis and nothing ever assumes lockstep; and because all three streams tick the same clock, any value can still be laid over any other exactly.



Six tiers on one master 10 µs clock, carried as three timestamp streams: 600/300/100, 50, and 10/1

Delta encoding and keyframes

Inside the file, data is stored by column, one field's entire time series at a time, so similar values sit together before the compressor sees them. Each column stores differences between consecutive samples rather than the values themselves, packed so that a small change costs a single byte, which is what sensor data mostly is. At regular intervals, every quarter second on the fast tiers, an absolute value is written instead of a difference. These keyframes bound how far an error can travel: a pure difference chain carries any bad value forward to the end of the column, whereas a keyframe restores the true value at the next interval. The timestamps themselves are protected the same way. Altitude values are stored to millimetre precision, finer than the sensor chain's own uncertainty, so no quantisation staircase is ever visible however far you zoom in.

Other devices

The Nano writes the same kind of file: same magic string, same LZMA container, same signing scheme, same columnar layout with zigzag varints, delta coding and keyframes. Everything in the two sections above about how values are packed applies unchanged. What differs is the shape.

A Nano records one table at one rate rather than six tiers, so there is no tier system and no separate timestamp streams: time is simply the first column, in milliseconds. The reserved header is 4 KB rather than 64 KB, and the columns begin at offset 4096. Where the Jupiter header holds t=0 and per-tier counts, the Nano header holds a product code, the sample count, the sample rate, the column count and the keyframe interval as consecutive 32-bit integers from offset 32, which means a decoder can read the column block of any Nano file without a version table, even though the meaning of each column is still version territory. The second half of the Jupiter header, the settings snapshot, is instead a block of plain CSV text at offset 256 carrying the flight summary: apogee, times, velocities, serial number, firmware and the settings string as flown. Nano files also carry a second Ed25519 signature, over the CSV the log expands to, so an exported CSV can be verified on its own.

Because both products share the magic string, a converter that accepts either has to look at the words after it. The Nano writes its product code at offset 36 and always has 15, 16 or 17 columns, recorded at offset 48; the Jupiter has t=0 at that offset and no product code at all. The reference converter below recognises the Nano positively on those two fields and treats anything else as a Jupiter.

Extracting the data yourself

First, to be clear: you never need to do any of this. Flights upload to altimetercloud.com, and every tool for viewing, analysing and converting your data lives there, so extraction by hand is entirely optional. A desktop application for Windows that opens

an .aclz and writes CSV directly, for both the Jupiter and the Nano, with no Python or command line involved, is in testing and will be released in the coming months. This section exists for the curious and for anyone building their own tooling.

The file itself is open to inspection, and this section is the map for doing it properly. Connect over USB and each flight is a flight_<t0>.aclz file, which is one standard LZMA stream: 7-Zip opens it directly, xz --decompress --format=lzma does on the command line, and Python's built-in lzma module does with format=lzma.FORMAT_ALONE.

Decompressed, a Jupiter file is a fixed 64 KB header followed by the data columns. The header fields you need for a decode all sit at fixed offsets: the ASCII magic string at offset 0; the format version as a 32-bit integer at offset 32, which you must check before touching the columns, because column counts and value scales change between versions and the data cannot tell you itself; t=0 at offset 36 as a 32-bit tick count on the 10 microsecond clock; the pad elevation as a float at 40; the six per-tier sample counts as 32-bit integers from offset 44; and the six per-tier keyframe intervals as 16-bit integers from offset 68. The settings snapshot begins at offset 32768: a SET1 marker, a 32-bit length, then key=value pairs separated by the 0x1F byte, close enough to plain text that a text editor shows it.

The columns begin at offset 65536 and are pure sequences of variable-length integers, one column completely before the next, in a fixed order: 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. Each value is a varint (seven bits per byte, low bits first, top bit meaning another byte follows) carrying a zigzag-encoded signed number, so decoding a value is (u >> 1) ^ -(u & 1). Reassembly is then the keyframe rule: within a column, sample i is an absolute value when i % kf == 0 for that tier's keyframe interval, and otherwise a delta to add to the previous decoded value. That is the entire mechanism. The NaN sentinel is INT32_MIN. Slower tiers within a group index into their group's timestamp stream (a 300 Hz sample j uses stream entry 2j, a 100 Hz sample j entry 6j, a 1 Hz sample j entry 10j of the third stream), and for format version 9 the field counts per tier are 3, 7, 7, 8, 18 and 49, not counting the timestamp streams themselves.

One detail worth getting right: the timestamps are unsigned 32-bit tick counts stored in signed columns, so subtract t=0 with a 32-bit wrap rather than plainly, or a device that has been powered for more than about six hours produces times that are wrong by billions. Samples from before launch legitimately decode as negative, because the pre-roll buffer was recorded before t=0.

The reference converter below is the canonical worked example, and it is the whole workflow in one command: run python3 aclz_to_csv.py flight_1234.aclz and decompression, header, settings and all six tiers come out with no other tools involved. It detects which product wrote the file and handles both: a Jupiter gives six tier CSVs plus settings.txt, a Nano gives one samples.csv plus its flight summary as metadata.csv. It decodes the header for any version, decodes the full Jupiter column set for version 9, and reads any Nano column count. It deliberately emits the raw quantised integers the device stored, since field meanings and scales are version territory; for CSVs in real units, use altimetercloud.com or the desktop application. It is also deliberately 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')
```

For a port, the decompress entry point plus three primitives, varint reading, zigzag decoding and delta-with-keyframe reassembly, are everything the format needs, and they are identical for every device we make; only the header offsets and the column table change. In Perl (note 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);
}
```

Compression and signature

The assembled file is compressed with LZMA at the effort chosen by the log_zlevel setting, covered on the Compression & Encryption page along with the signing scheme's strength. The columnar delta layout and the compressor work together: the deltas flatten the data, the columns group it, and LZMA removes what remains. The signature underneath makes the result permanent evidence: your flight, as flown, provably unmodified.