Phase 2: Implement FQZComp Encoding — Detailed Plan

Overview

FQZComp is a context-adaptive quality score compressor that uses an arithmetic range coder with up to 65,536 adaptive frequency models. It is the single biggest engineering effort remaining for full CRAM 3.1 write support. Without FQZComp encoding, htsjdk cannot offer the `small` or `archive` profiles (both use fqzcomp for `QS_QualityScore`).

The decoder (`FQZCompDecode.java`) is fully implemented and serves as the authoritative reference for the wire format. The encoder must produce output that `FQZCompDecode.uncompress()` can consume. The low-level encoding primitives (`ByteModel.modelEncode()`, `RangeCoder.rangeEncode()`, `RangeCoder.rangeEncodeEnd()`) already exist and are tested — the gap is purely in the orchestration and parameter selection.

Assumes: Phase 1 merged (`cn_cram_3_1_write` branch), so htsjdk writes CRAM 3.1 with RANSNx16 and Name Tokeniser.

Architecture Overview

The FQZComp encoder is a two-pass process:

1. **Analysis pass**: Scan the quality data to determine optimal encoding parameters (number of distinct quality values, whether reads are fixed-length, whether to enable position/delta/selector context, quality remapping table, etc.)
2. **Encoding pass**: Serialize the parameters, then encode each quality byte using the adaptive `ByteModel` array indexed by a 16-bit context that updates per-symbol.

The encoder must produce this wire format (matching what `FQZCompDecode.uncompress()` reads):

```
[uint7]     uncompressed size
[1 byte]    version (must be 5)
[FQZParams] parameter header (global flags, param blocks, tables)
[...]       range-coded quality data (adaptive arithmetic bitstream)
[5 bytes]   range coder flush (rangeEncodeEnd)
```

Task 1: Implement FQZUtils.writeArray() — Table Serialization

File: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZUtils.java`

The decoder has `FQZUtils.readArray(ByteBuffer, int[], int)` which reads RLE-encoded lookup tables (qtab, ptab, dtab) from the stream. The encoder needs the inverse: `writeArray(ByteBuffer, int[], int)`.

Current readArray format (from decoder):

```
For each entry in the array:
  [1 byte] value (the table entry, 0-255)
  If the next byte's high bit is set:
    [1 byte] run length = (byte & 0x7F), meaning repeat this value for
`run` more entries
  Otherwise:
    advance to next entry (run of 1)
```

What to implement:

```
public static void writeArray(final ByteBuffer outBuffer, final int[]
array, final int size) {
    for (int i = 0; i < size; i++) {
        outBuffer.put((byte) array[i]);
        // Count how many subsequent entries have the same value
        int run = 0;
        while (i + run + 1 < size && array[i + run + 1] == array[i] && run
< 127) {
            run++;
        }
        if (run > 0) {
            outBuffer.put((byte) (run | 0x80));
            i += run;
        }
    }
}
```

Test: Round-trip: `writeArray` → `readArray` should produce identical arrays. Test with uniform arrays, alternating values, runs of varying lengths, and arrays of length 256 and 1024.

Task 2: Implement FQZParam Serialization (Write Side)

File: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZParam.java`

Currently `FQZParam` has only a read-from-stream constructor. Add a write method and a programmatic constructor.

2a: Add a programmatic constructor

```
public FQZParam(
    int context,
    int parameterFlags,        // DEDUP | FIXED_LEN | SEL | QMAP | PTAB |
DTAB | QTAB
    int maxSymbols,
    int qualityContextBits,
    int qualityContextShift,
    int qualityContextLocation,
    int selectorContextLocation,
    int positionContextLocation,
    int deltaContextLocation,
    int[] qualityMap,         // null if !doQmap, else size maxSymbols
    int[] qualityContextTable,// null if !doQtab, else size 256
    int[] positionContextTable,// null if !doPos, else size 1024
    int[] deltaContextTable   // null if !doDelta, else size 256
)
```

2b: Add a write method

```
public void write(final ByteBuffer outBuffer) {
    // 2 bytes: context
    outBuffer.put((byte) (context & 0xFF));
    outBuffer.put((byte) ((context >> 8) & 0xFF));
    // 1 byte: flags
    outBuffer.put((byte) buildParameterFlags());
    // 1 byte: maxSymbols
    outBuffer.put((byte) maxSymbols);
    // 1 byte: x = (qualityContextBits << 4) | qualityContextShift
    outBuffer.put((byte) ((qualityContextBits << 4) |
qualityContextShift));
    // 1 byte: y = (qualityContextLocation << 4) | selectorContextLocation
    outBuffer.put((byte) ((qualityContextLocation << 4) |
selectorContextLocation));
    // 1 byte: z = (positionContextLocation << 4) | deltaContextLocation
    outBuffer.put((byte) ((positionContextLocation << 4) |
deltaContextLocation));
    // optional tables
    if (doQmap) writeQualityMap(outBuffer);
    if (doQtab) FQZUtils.writeArray(outBuffer, qualityContextTable, 256);
    if (doPos)  FQZUtils.writeArray(outBuffer, positionContextTable,
1024);
    if (doDelta) FQZUtils.writeArray(outBuffer, deltaContextTable, 256);
}
```

Wire format reference (from the read constructor):

```
[2 bytes LE] context (base context offset)
[1 byte]     parameterFlags
[1 byte]     maxSymbols
[1 byte]     x = (qualityContextBits << 4) | qualityContextShift
[1 byte]     y = (qualityContextLocation << 4) | selectorContextLocation
[1 byte]     z = (positionContextLocation << 4) | deltaContextLocation
[if QMAP]    qualityMap[maxSymbols] (raw bytes, 1 per symbol)
[if QTAB]    qualityContextTable[256] (RLE-encoded via writeArray)
```

```
[if PTAB]    positionContextTable[1024] (RLE-encoded via writeArray)
[if DTAB]    deltaContextTable[256] (RLE-encoded via writeArray)
```

Test: Round-trip: construct FQZParam programmatically → write → read back → compare all fields.

Task 3: Implement FQZParams Serialization (Write Side)

File: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZParams.java`

3a: Add a programmatic constructor

```
public FQZParams(
    FQZGlobalFlags globalFlags,
    List<FQZParam> paramList,
    int maxSelector,
    int[] selectorTable    // null if !hasSelectorTable, else size 256
)
```

3b: Add a write method

```
public void write(final ByteBuffer outBuffer) {
    // 1 byte: global flags
    globalFlags.write(outBuffer);
    // if multiParam: 1 byte numParamBlocks
    if (globalFlags.isMultiParam()) {
        outBuffer.put((byte) fqzParamList.size());
    }
    // if hasSelectorTable: 1 byte maxSelector, then RLE-encoded
selectorTable[256]
    if (globalFlags.hasSelectorTable()) {
        outBuffer.put((byte) maxSelector);
        FQZUtils.writeArray(outBuffer, selectorTable, 256);
    }
    // write each param block
    for (FQZParam param : fqzParamList) {
        param.write(outBuffer);
    }
}
```

3c: Add FQZGlobalFlags.write()

File: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZGlobalFlags.java`

Add a `write(ByteBuffer)` method and a programmatic constructor. The global flags byte has bits for: `MULTI_PARAM`, `SELECTOR_TABLE`, `DO_REVERSE`.

Test: Round-trip: construct FQZParams → write → read back → compare all fields including selectorTable and all child FQZParam objects.

Task 4: Implement Parameter Selection / Presets

New file: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZPresets.java`

This is the most intellectually challenging part. Given raw quality score data and per-read metadata, the encoder must choose optimal FQZParam values. htslib's `htscodecs` library provides multiple presets (`fqz_preset`, levels -1 through -9).

Minimal viable implementation (single preset, equivalent to htslib's default):

The simplest approach is a single "normal" preset that:

1. **Scans quality values** to find `maxSymbols` (the number of distinct quality values present)
2. **Builds a quality map** (`qmap`) that remaps the quality alphabet to a contiguous range `0..maxSymbols-1` for better model efficiency
3. **Builds a quality context table** (`qtab`) that bins quality values into a small number of context categories (e.g., 4-8 bins)
4. **Sets context parameters**:
   - `qualityContextBits = 2` (use 2 bits from quality context → 4 quality contexts)
   - `qualityContextShift = 1` (shift quality context left by 1 per position)
   - `qualityContextLocation = 0` (quality context occupies bits 0-1 of the 16-bit context)
   - `positionContextLocation` and `deltaContextLocation` set based on whether position/delta are enabled
5. **Decides which features to enable**:
   - `doQmap`: yes, if maxSymbols < 42 (reduces model size)
   - `doQtab`: yes (always beneficial for quality context compression)
   - `doPos`: optional (helps for position-dependent quality patterns)
   - `doDelta`: optional (helps when quality degrades along reads)
   - `doDedup`: optional (helps when many reads share identical quality strings)
   - `doSel`: no (single param block for minimal implementation)
   - `fixedLen`: set if all reads are the same length

Reference implementations:

- **htslib/htscodecs** `fqzcomp_qual.c`: `fqz_manual_parameters()` and the preset arrays `fqz_preset[]`. The C code has 10 preset levels with different context configurations.
- **noodles** `fqzcomp/encode.rs`: Rust implementation with similar parameter selection.

Recommended approach:

Start with **one hardcoded preset** equivalent to htslib's `-3` level (the default for `normal` profile). This uses:

- Quality context: 2 bits, shift 1
- Position context: disabled
- Delta context: disabled

- Quality map: enabled
- Quality context table: enabled (identity or simple binning)
- Single param block (no selector)

This produces valid, interoperable FQZComp output. Additional presets can be added later for `small` and `archive` profiles.

```
public class FQZPresets {
    public static FQZParams buildDefaultParams(byte[] qualityData, int[]
readLengths) {
        // 1. Analyze quality alphabet
        int maxQuality = 0;
        boolean[] seen = new boolean[256];
        for (byte q : qualityData) { seen[q & 0xFF] = true; maxQuality =
Math.max(maxQuality, q & 0xFF); }
        int maxSymbols = maxQuality + 1;

        // 2. Build quality map (compress alphabet)
        int[] qmap = buildQualityMap(seen, maxSymbols);

        // 3. Build quality context table (bin qualities into context
categories)
        int[] qtab = buildQualityContextTable(maxSymbols);

        // 4. Check fixed-length reads
        boolean fixedLen = allSameLength(readLengths);

        // 5. Construct FQZParam with default context configuration
        FQZParam param = new FQZParam(
            /* context= */ 0,
            /* flags= */ QMAP_FLAG | QTAB_FLAG | (fixedLen ?
FIXED_LEN_FLAG : 0),
            /* maxSymbols= */ maxSymbols,
            /* qualityContextBits= */ 2,
            /* qualityContextShift= */ 1,
            /* qualityContextLocation= */ 0,
            /* selectorContextLocation= */ 0,
            /* positionContextLocation= */ 0,
            /* deltaContextLocation= */ 0,
            qmap, qtab, null, null
        );

        // 6. Wrap in FQZParams (single param block, no selector)
        return new FQZParams(
            new FQZGlobalFlags(0),  // no multi-param, no selector table,
no reverse
            List.of(param),
            /* maxSelector= */ 0,
            null
        );
    }
}
```

Task 5: Implement FQZCompEncode.compress()

File: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZCompEncode.java`

This is the core encoding method. It mirrors `FQZCompDecode.uncompress()` exactly, using `modelEncode` instead of `modelDecode`.

5a: Update the compress() signature

The current signature is `compress(ByteBuffer inBuffer)`. It needs access to per-read metadata. Two options:

Option A (recommended): Add an overloaded method that accepts read lengths:

```
public ByteBuffer compress(final ByteBuffer inBuffer, final int[]
readLengths, final int numReads)
```

Option B: Keep the single-buffer signature and detect read boundaries from the data (requires knowing read lengths from external context). This is less clean but avoids changing the interface.

5b: Implement the encoding loop

```
public ByteBuffer compress(final ByteBuffer inBuffer, final int[]
readLengths, final int numReads) {
    final int uncompressedSize = inBuffer.remaining();

    // 1. Build parameters from analysis of quality data
    final FQZParams fqzParams = FQZPresets.buildDefaultParams(
        toByteArray(inBuffer), readLengths);

    // 2. Allocate output buffer (generous: uncompressed size + header
overhead)
    final ByteBuffer outBuffer =
CompressionUtils.allocateOutputBuffer(uncompressedSize);

    // 3. Write header
    CompressionUtils.writeUint7(outBuffer, uncompressedSize);
    outBuffer.put((byte) SUPPORTED_FQZCOMP_VERSION);  // version = 5
    fqzParams.write(outBuffer);

    // 4. Initialize models and state (identical to decoder)
    final FQZModels fqzModels = new FQZModels(fqzParams);
    final FQZState fqzState = new FQZState();
    final RangeCoder rangeCoder = new RangeCoder();
    // Note: no rangeEncodeStart — the range coder starts implicitly for
encoding
```

```
    // 5. Encode each quality byte
    int readIndex = 0;
    int posInRead = 0;
    FQZParam fqzParam = null;
    int last = 0;

    for (int i = 0; i < uncompressedSize; ) {
        fqzState.resizeArrays(fqzState.getReadOrdinal());

        if (fqzState.getBases() == 0) {
            // Start of a new read — encode per-read header
            fqzState.setContext(0);
            encodeFQZNewRecord(outBuffer, rangeCoder, fqzModels,
fqzParams, fqzState,
                               readLengths[readIndex]);

            if (fqzState.getIsDuplicate()) {
                // Copy previous read's qualities (already in output)
                i += fqzState.getRecordLength();
                fqzState.setBases(0);
                readIndex++;
                continue;
            }

            fqzParam =
fqzParams.getFQZParamList().get(fqzState.getSelectorTable());
            last = fqzState.getContext();
        }

        // Get the quality value to encode
        int quality = inBuffer.get(i) & 0xFF;

        // Apply quality map (binning) if enabled — reverse of decode's
unbinning
        int compressedQuality = quality;
        if (fqzParam.isDoQmap()) {
            compressedQuality = fqzParam.getReverseQualityMap()[quality];
        }

        // Encode one quality symbol using context `last`
        fqzModels.getQuality()[last].modelEncode(outBuffer, rangeCoder,
compressedQuality);

        // Update context for next symbol (identical to decoder)
        last = FQZCompDecode.fqzUpdateContext(fqzParam, fqzState,
compressedQuality);
        fqzState.setContext(last);
        i++;
    }

    // 6. Flush the range coder
    rangeCoder.rangeEncodeEnd(outBuffer);

    outBuffer.limit(outBuffer.position());
    outBuffer.rewind();
    return outBuffer;
}
```

5c: Implement encodeFQZNewRecord()

Mirrors `decodeFQZNewRecord()` — encodes per-read header data:

```
private static void encodeFQZNewRecord(
        final ByteBuffer outBuffer,
        final RangeCoder rangeCoder,
        final FQZModels models,
        final FQZParams fqzParams,
        final FQZState state,
        final int readLength) {

    // 1. Encode selector (which param block)
    if (fqzParams.getMaxSelector() > 0) {
        models.getSelector().modelEncode(outBuffer, rangeCoder,
state.getSelector());
    } else {
        state.setSelector(0);
    }
    state.setSelectorTable(fqzParams.getSelectorTable()
[state.getSelector()]);
    final FQZParam params =
fqzParams.getFQZParamList().get(state.getSelectorTable());

    // 2. Encode read length (4 bytes LE, each via separate ByteModel)
    if (params.getFixedLen() >= 0) {
        models.getLength()[0].modelEncode(outBuffer, rangeCoder,
readLength & 0xFF);
        models.getLength()[1].modelEncode(outBuffer, rangeCoder,
(readLength >> 8) & 0xFF);
        models.getLength()[2].modelEncode(outBuffer, rangeCoder,
(readLength >> 16) & 0xFF);
        models.getLength()[3].modelEncode(outBuffer, rangeCoder,
(readLength >> 24) & 0xFF);
        if (params.getFixedLen() > 0) params.setFixedLen(-readLength);
    }
    // else: fixedLen < 0 means length already known, nothing to encode

    state.setRecordLength(readLength);

    // 3. Encode reverse flag if needed
    if (fqzParams.getFQZFlags().doReverse()) {
        models.getReverse().modelEncode(outBuffer, rangeCoder,
            state.getReverseArray()[state.getReadOrdinal()] ? 1 : 0);
    }

    // 4. Encode duplicate flag if needed
    state.setIsDuplicate(false);
    if (params.isDoDedup()) {
        boolean isDup = checkDuplicate(state);  // compare with previous
read's qualities
        models.getDuplicate().modelEncode(outBuffer, rangeCoder, isDup ? 1
: 0);
        if (isDup) state.setIsDuplicate(true);
    }

    // 5. Reset per-record state
    state.setBases(readLength);
    state.setDelta(0);
    state.setQualityContext(0);
    state.setPreviousQuality(0);
    state.setReadOrdinal(state.getReadOrdinal() + 1);
}
```

5d: Add reverse quality map to FQZParam

The decoder uses `qualityMap[compressedSymbol] → originalQuality` to unbinned. The encoder needs the inverse: `reverseQualityMap[originalQuality] → compressedSymbol`. Add:

```
// In FQZParam:
private int[] reverseQualityMap;  // size 256, maps original quality →
compressed symbol

// Built from qualityMap:
public void buildReverseQualityMap() {
    reverseQualityMap = new int[256];
    for (int i = 0; i < maxSymbols; i++) {
        reverseQualityMap[qualityMap[i]] = i;
    }
}
```

Task 6: Wire FQZCompExternalCompressor into the Write Path

File: `src/main/java/htsjdk/samtools/cram/compression/fqzcomp/FQZCompExternalCompressor.java`

6a: Implement compress()

Replace the `throw new UnsupportedOperationException(...)` with:

```
@Override
public byte[] compress(byte[] data, final CRAMCodecModelContext
contextModel) {
    final int[] readLengths = contextModel.getReadLengths();
    final int numReads = contextModel.getNumReads();
    return CompressionUtils.toByteArray(
        fqzCompEncoder.compress(CompressionUtils.wrap(data), readLengths,
numReads));
}
```

6b: Store the encoder reference

Change the constructor to actually store the encoder:

```
public FQZCompExternalCompressor(
        final FQZCompEncode fqzCompEncoder,
        final FQZCompDecode fqzCompDecoder) {
    super(BlockCompressionMethod.FQZCOMP);
    this.fqzCompEncoder = fqzCompEncoder;   // was unused_, now used
    this.fqzCompDecoder = fqzCompDecoder;
}
```

Task 7: Populate CRAMCodecModelContext

File: `src/main/java/htsjdk/samtools/cram/structure/CRAMCodecModelContext.java`

This class is currently empty. It needs to carry per-slice read metadata for FQZComp.

7a: Add fields

```
public class CRAMCodecModelContext {
    private int[] readLengths;      // length of each read in this slice
    private int numReads;           // number of reads in this slice
    // Future: boolean[] isDuplicate, boolean[] isReverse for advanced
presets

    public int[] getReadLengths() { return readLengths; }
    public void setReadLengths(int[] readLengths) { this.readLengths =
readLengths; }
    public int getNumReads() { return numReads; }
    public void setNumReads(int numReads) { this.numReads = numReads; }
}
```

7b: Populate the context in the write path

The context is created in `Slice.java` and passed through to `CramRecordWriter.writeToSliceBlocks()`. The read lengths are available from the `CRAMCompressionRecord` objects being written. The population point is in the slice write path, likely in `CramRecordWriter` or `SliceFactory`, before `flushStreamsToBlocks(contextModel)` is called.

File: `src/main/java/htsjdk/samtools/cram/build/CramRecordWriter.java` (or wherever records are iterated before compression)

```
// Before flushing streams to blocks:
int[] readLengths = new int[records.size()];
for (int i = 0; i < records.size(); i++) {
    readLengths[i] = records.get(i).getReadLength();
}
contextModel.setReadLengths(readLengths);
contextModel.setNumReads(records.size());
```

Important: Other compressors (`GZIP`, `RANSNx16`, `NameTokeniser`) must continue to work with null/empty context. Their `compress(data, contextModel)` methods already accept and ignore the context parameter.

Task 8: Integrate FQZComp into Encoding Strategy for Small/Archive Profiles

File: `src/main/java/htsjdk/samtools/cram/structure/CompressionHeaderEncodingMap.java`

This task ties into the profile system (see Phase 3 plan for the full profile design). For this phase, the minimal integration is:

8a: Add a helper method for FQZComp encoding of quality scores

```
private void putExternalFQZCompEncoding(final DataSeries dataSeries) {
    putExternalEncoding(
        dataSeries,

compressorCache.getCompressorForMethod(BlockCompressionMethod.FQZCOMP,
0));
}
```

8b: Make quality score codec configurable

When the profile is `SMALL` or `ARCHIVE`, the `QS_QualityScore` data series should use FQZComp instead of RANSNx16:

```
// In initializeDefaultEncodings() or a profile-aware variant:
if (profile == EncodingProfile.SMALL || profile ==
EncodingProfile.ARCHIVE) {
    putExternalFQZCompEncoding(DataSeries.QS_QualityScore);
} else {
    putExternalRansOrderOneEncoding(DataSeries.QS_QualityScore);
}
```

8c: Add FQZComp to getBestExternalCompressor() for quality-score blocks

For `small`/`archive` profiles, `getBestExternalCompressor()` should also try FQZComp as a candidate when compressing tag data that contains quality-like values. However, FQZComp requires the `CRAMCodecModelContext` (read lengths), so it can only be used for the `QS_QualityScore` data series where context is available — not for arbitrary tag blocks. This means FQZComp should NOT be added to `getBestExternalCompressor()` (which is used for tags) but should only be wired through the fixed data-series assignment as in 8b.

Task 9: Tests

9a: FQZUtils.writeArray() round-trip tests

File: `src/test/java/htsjdk/samtools/cram/compression/fqzcomp/` (new or extend existing)

- Test with uniform arrays, random arrays, edge cases (single element, max-length runs)
- Verify `writeArray()` → `readArray()` produces identical data

9b: FQZParam / FQZParams serialization round-trip tests

- Construct params programmatically → write to buffer → read back → assert equality
- Test with various flag combinations (qmap on/off, qtab on/off, ptab on/off, dtab on/off)
- Test multi-param configs with selector tables

9c: FQZCompEncode round-trip tests (encode → decode)

- Encode quality data with `FQZCompEncode.compress()` → decode with `FQZCompDecode.uncompress()` → compare
- Test with uniform quality (all Q30), random quality, short reads, long reads, variable-length reads, fixed-length reads
- Test with different preset configurations

9d: Interop tests against htscodecs test data

Directory: `src/test/resources/htsjdk/hts-specs/test/cram/codecs/fqzcomp/`

There are 16 existing test files (`q4.0` through `qvar.3`). Currently `FQZCompInteropTest.java` only tests decode. Extend with:

- Decode each test file, then re-encode with `FQZCompEncode`, then decode again — verify the round-trip produces identical quality data
- Note: the re-encoded bitstream will NOT be byte-identical to htscodecs output (different parameter choices, different model evolution). Only the decoded quality data must match.

9e: End-to-end CRAM 3.1 write tests with FQZComp

- Write a CRAM 3.1 file with FQZComp enabled for `QS_QualityScore`
- Read back with htsjdk — verify quality scores match
- Read with samtools — verify interop (requires samtools on path)

9f: Update StructureTestUtils exclusion

File: `src/test/java/htsjdk/samtools/cram/structure/StructureTestUtils.java`

Remove `BlockCompressionMethod.FQZCOMP` from the compressor exclusion set now that encoding is implemented.

Task 10: Performance Considerations

10a: Memory usage

`FQZModels` allocates **65,536 `ByteModel` instances**, each containing a frequency array of size `maxSymbols + 1`. For typical quality data with ~42 distinct values, this is `65,536 × 43 × 4 bytes ≈ 11 MB` of model state. This is acceptable for server-side use but should be documented.

10b: Encoding speed

`ByteModel.modelEncode()` iterates linearly through the symbol list to find the cumulative frequency. For `maxSymbols ≈ 42`, this is ~21 iterations on average per symbol. This is the same cost as decoding, which is already accepted. No optimization needed for Phase 2.

10c: Output buffer sizing

The encoder should allocate an output buffer of at least `uncompressedSize + headerOverhead` bytes. If the compressed output exceeds the uncompressed size (possible for high-entropy quality data), the encoder should fall back to RANSNx16 (not FQZComp). This fallback should be handled at the `CompressionHeaderEncodingMap` level, not inside `FQZCompEncode`.

What Phase 2 Does NOT Change

- **Stripe encoding** — remains unimplemented (Phase 3)
- **Multiple FQZComp presets** — only one default preset is implemented. Additional presets for `small` vs `archive` quality levels are a future enhancement.
- **Duplicate detection in FQZComp** — `doDedup` is not enabled in the default preset (requires comparing quality strings between consecutive reads, which adds complexity). Can be added later.
- **Reverse quality encoding** — `doReverse` is not enabled in the default preset. Can be added later.
- **CRAMEncodingStrategy profile enum** — Phase 2 adds the FQZComp codec but does not add the profile selection UI. The profile system is covered in Phase 3.
- **Position and delta context** — Not enabled in the default preset. These improve compression for position-dependent quality patterns but add complexity.

Dependencies

- **Phase 1 must be merged** (CRAM 3.1 write infrastructure in place)
- **No external library dependencies** — all encoding primitives (`ByteModel.modelEncode()`, `RangeCoder`) already exist

- **CRAMCodecModelContext** changes (Task 7) touch the write path broadly but are backward-compatible (all existing compressors ignore the context)

Risk Assessment

Task	Effort	Risk	Notes
Task 1 (writeArray)	Low	Low	Straightforward inverse of readArray
Task 2-3 (param serialization)	Medium	Low	Wire format is well-defined by decoder
Task 4 (presets)	High	Medium	Parameter selection is the creative/empirical part
Task 5 (compress loop)	High	Medium	Must exactly mirror decoder's state machine
Task 6 (external compressor)	Low	Low	Wiring only
Task 7 (context model)	Medium	Medium	Touches the write path broadly
Task 8 (encoding map)	Low	Low	Conditional codec assignment
Task 9 (tests)	Medium	Low	Many test dimensions

Total estimated scope: ~1000-1500 lines of new code across 8-10 files.

Highest risk: Getting the encoder state machine exactly right so that `encode → decode` produces identical output. The decoder is the ground truth. Any divergence in context update order, model initialization, or parameter serialization will produce corrupt output. Extensive round-trip testing is essential.