# Transformer — Zero to Hero: Complete Technical Manual

Exhaustive explanation, diagrams, numeric examples, and PyTorch-style appendix

Intended coverage: fundamentals → attention → multi-head → residuals → FFN → layernorm → pos enc → KV cache → training/decoding → implementation notes

## How to use this manual

This document is structured to teach every concept from first principles, then present the exact formulas and implementation notes you need to write bug-free code. Key results are boxed. Diagrams are simplified for robustness and clarity. A numerical worked example shows attention computed step-by-step (digit-by-digit) to remove any ambiguity.

## 1 Prerequisites: linear algebra and probability refresher

We assume basic familiarity with vectors, matrices and common operations. Quick recap:

- A vector $v \in \mathbb{R}^d$ is a column of $d$ numbers. A matrix $A \in \mathbb{R}^{m\times n}$ maps $\mathbb{R}^n \to \mathbb{R}^m$ via $Ax$.
- Matrix multiplication: $(AB)_{ij} = \sum_k A_{ik}B_{kj}$.
- Transpose: $A^\top$ flips rows/columns.
- Dot product: for $u, v \in \mathbb{R}^d$, $u \cdot v = u^\top v = \sum_{i=1}^d u_i v_i$.
- Softmax (row-wise): for vector $z \in \mathbb{R}^n$,

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}.$$

  Numerically stable trick: subtract the max before exponentiating: $\text{softmax}(z) = \text{softmax}(z - \max(z))$.

**Notation used throughout:**

- $N$ — sequence length (tokens).
- $d$ — model (embedding) dimension.
- $H$ — number of attention heads.
- $d_{\text{head}}$ — per-head dimensionality (so typically $d = H \cdot d_{\text{head}}$).
- $d_{ff}$ — hidden dimension of the position-wise feedforward network.
- $X \in \mathbb{R}^{N\times d}$ — token embeddings for a full sequence.

## 2 High-level architecture

A single (decoder-only) Transformer layer block processes token embeddings through:

LayerNorm → MultiHeadAttention (masked) → Residual add → LayerNorm → FFN → Residual add.

In equation (pre-layernorm variant):

$$y_1 = x + \text{MultiHead}(\text{LayerNorm}(x)), \qquad y = y_1 + \text{FFN}(\text{LayerNorm}(y_1)).$$

### 2.1 What each piece does (plain English)

- **Embedding**: map token IDs to vectors.
- **Positional encoding**: inject token position information into embeddings.
- **Multi-head attention**: each token can "read" information from other tokens via learned queries/keys/values.
- **Residual connections**: add the input to outputs to preserve low-level signal and ease training.
- **LayerNorm**: normalize vector components per token for stable optimization.
- **FFN**: position-wise MLP to increase representation power.

## 3 Attention: full derivation and implementation

### 3.1 Intuition

A token forms a *query* that asks "what I want to fetch from the context". Each context token exposes a *key* describing what it contains and a *value* describing what to pass on. Attention scores measure query-key similarity; softmax weights values to produce the output.

### 3.2 Scaled dot-product attention (single head)

Given $X \in \mathbb{R}^{N \times d}$. Learn three projections:

$$W_Q, W_K, W_V \in \mathbb{R}^{d \times d_{\text{head}}}.$$

Compute:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

so $Q, K, V \in \mathbb{R}^{N \times d_{\text{head}}}$.

Scores matrix (unnormalized):

$$S = QK^\top \in \mathbb{R}^{N \times N}, \qquad S_{ij} = Q_i \cdot K_j.$$

Scale and add optional mask $M$:

$$\tilde{S} = \frac{S}{\sqrt{d_{\text{head}}}} + M.$$

Row-wise softmax:

$$A = \text{softmax}(\tilde{S}) \quad \text{(apply softmax to each row of } \tilde{S}\text{)}.$$

Attention output:

$$\text{Attention}(Q, K, V) = AV \in \mathbb{R}^{N \times d_{\text{head}}}.$$

**Why scale by $\sqrt{d_{\text{head}}}$?** Dot products grow in magnitude with dimension; dividing stabilizes the softmax input, preventing extreme probabilities that kill gradients.

### 3.3 Mask $M$ for autoregressive decoding

For decoder-only models (causal LM), future positions must be blocked:

$$M_{ij} = \begin{cases} 0 & j \le i, \\ -\infty & j > i. \end{cases}$$

In practice, use a large negative constant (e.g., $-1e9$) for numerical compatibility.

## 4 Multi-head attention: math and shapes

- For each head $h = 1, \ldots, H$, have independent projections $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{d \times d_{\text{head}}}$.
- Compute head outputs: $\text{head}_h = \text{Attention}(XW_Q^{(h)}, XW_K^{(h)}, XW_V^{(h)}) \in \mathbb{R}^{N \times d_{\text{head}}}$.
- Concatenate heads: $[\text{head}_1 \| \cdots \| \text{head}_H] \in \mathbb{R}^{N \times (Hd_{\text{head}})}$.
- Final linear projection: multiply by $W_O \in \mathbb{R}^{Hd_{\text{head}} \times d}$ to produce $\mathbb{R}^{N \times d}$.

## 5 Residual stream and layer normalization

### 5.1 Residual connections

If a sub-layer computes $F(x)$, the residual update is $x \leftarrow x + F(x)$. This keeps identity paths and makes gradient flow easier.

### 5.2 Layer Normalization (LayerNorm)

For a token vector $x \in \mathbb{R}^d$:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2 + \epsilon},$$

$$\text{LayerNorm}(x) = \gamma \frac{x - \mu}{\sigma} + \beta,$$

with learnable $\gamma, \beta \in \mathbb{R}^d$.
**Pre-norm vs post-norm:** apply LayerNorm before sublayers (pre-norm) for improved stability in deep models. Many modern implementations use pre-norm.

## 6 Position-wise Feedforward Network (FFN)

Function applied independently to each position:

$$\text{FFN}(x) = W_2\, \sigma(W_1 x + b_1) + b_2.$$

Typical: $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, with $d_{ff} \approx 4d$ typical.
Activations: GeLU or SwiGLU are common and perform better than ReLU in practice.

## 7 Token embeddings and positional encodings

- **Token Embedding matrix** $E \in \mathbb{R}^{|V| \times d}$. Token ids $\rightarrow$ row vectors.
- **Positional Encoding** options:
  1. Learned positional embeddings $P \in \mathbb{R}^{L \times d}$ (where $L$ is maximum length), added: $X \leftarrow X + P$.
  2. Sinusoidal encodings (original): deterministic functions of position.

## 8 KV cache for fast autoregressive inference

At inference time, recomputing $K, V$ for all previous tokens every step is wasteful. Cache $K$ and $V$ per layer and head, append new $K, V$ at each step, then attention uses cached $K, V$ plus current $Q$. This reduces compute from $O(N^2)$ to $O(N)$ per generated token at each step for the newly computed projections (but memory increases as $O(N)$ per layer).

## 9 Numerical worked example: attention computed step-by-step

We show a full numeric example, small and explicit. All arithmetic is listed.

### Setup

Choose:

$$N = 3 \quad (3 \text{ tokens}), \qquad d = 4, \qquad d_{\text{head}} = 2, \quad H = 1 \text{ (single head for clarity)}.$$

Input embeddings $X \in \mathbb{R}^{3 \times 4}$:

$$X = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Projection matrices (small integer values for clarity):

$$W_Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}, \qquad W_K = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad W_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

### Step 1: compute $Q = XW_Q$

Compute each row of $Q$ as $X_i \cdot W_Q$.
Row 1: $X_1 = (1, 0, 2, 1)$.

$$Q_{1,1} = 1 \cdot 1 + 0 \cdot 0 + 2 \cdot 1 + 1 \cdot 0 = 1 + 0 + 2 + 0 = 3.$$

$$Q_{1,2} = 1 \cdot 0 + 0 \cdot 1 + 2 \cdot 1 + 1 \cdot 1 = 0 + 0 + 2 + 1 = 3.$$

So row 1 of $Q$ is $(3, 3)$.
Row 2: $X_2 = (0, 1, 0, 1)$.

$$Q_{2,1} = 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 = 0.$$

$$Q_{2,2} = 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 1 = 0 + 1 + 0 + 1 = 2.$$

So row 2: $(0, 2)$.

Row 3: $X_3 = (1, 1, 1, 0)$.

$$Q_{3,1} = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 = 1 + 0 + 1 + 0 = 2.$$

$$Q_{3,2} = 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 1 = 0 + 1 + 1 + 0 = 2.$$

Row 3: $(2, 2)$.

Thus

$$Q = \begin{pmatrix} 3 & 3 \\ 0 & 2 \\ 2 & 2 \end{pmatrix}.$$

**Step 2: compute $K = XW_K$**

Row 1: $X_1 = (1, 0, 2, 1)$.

$$K_{1,1} = 1 \cdot 0 + 0 \cdot 1 + 2 \cdot 1 + 1 \cdot 0 = 0 + 0 + 2 + 0 = 2.$$

$$K_{1,2} = 1 \cdot 1 + 0 \cdot 0 + 2 \cdot 0 + 1 \cdot 1 = 1 + 0 + 0 + 1 = 2.$$

Row 1: $(2, 2)$.

Row 2: $X_2 = (0, 1, 0, 1)$.

$$K_{2,1} = 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0 = 0 + 1 + 0 + 0 = 1.$$

$$K_{2,2} = 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 = 0 + 0 + 0 + 1 = 1.$$

Row 2: $(1, 1)$.

Row 3: $X_3 = (1, 1, 1, 0)$.

$$K_{3,1} = 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 = 0 + 1 + 1 + 0 = 2.$$

$$K_{3,2} = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 + 0 \cdot 1 = 1 + 0 + 0 + 0 = 1.$$

Row 3: $(2, 1)$.

So

$$K = \begin{pmatrix} 2 & 2 \\ 1 & 1 \\ 2 & 1 \end{pmatrix}.$$

**Step 3: compute $V = XW_V$**

Row 1: $X_1 = (1, 0, 2, 1)$.

$$V_{1,1} = 1 \cdot 1 + 0 \cdot 0 + 2 \cdot 0 + 1 \cdot 1 = 1 + 0 + 0 + 1 = 2.$$

$$V_{1,2} = 1 \cdot 0 + 0 \cdot 1 + 2 \cdot 1 + 1 \cdot 0 = 0 + 0 + 2 + 0 = 2.$$

Row 1: $(2, 2)$.

Row 2: $X_2 = (0, 1, 0, 1)$.

$$V_{2,1} = 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 = 0 + 0 + 0 + 1 = 1.$$

$$V_{2,2} = 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0 = 0 + 1 + 0 + 0 = 1.$$

Row 2: $(1, 1)$.

Row 3: $X_3 = (1, 1, 1, 0)$.

$$V_{3,1} = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 + 0 \cdot 1 = 1 + 0 + 0 + 0 = 1.$$

$$V_{3,2} = 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 = 0 + 1 + 1 + 0 = 2.$$

Row 3: $(1, 2)$.

So

$$V = \begin{pmatrix} 2 & 2 \\ 1 & 1 \\ 1 & 2 \end{pmatrix}.$$

### Step 4: compute score matrix $S = QK^\top$

We compute $S_{ij} = Q_i \cdot K_j$.

Row 1 dot row 1:

$$S_{11} = (3,3) \cdot (2,2) = 3 \cdot 2 + 3 \cdot 2 = 6 + 6 = 12.$$

Row 1 dot row 2:

$$S_{12} = (3,3) \cdot (1,1) = 3 + 3 = 6.$$

Row 1 dot row 3:

$$S_{13} = (3,3) \cdot (2,1) = 3 \cdot 2 + 3 \cdot 1 = 6 + 3 = 9.$$

So first row of $S$: $(12, 6, 9)$.

Row 2: $Q_2 = (0,2)$.

$$S_{21} = (0,2) \cdot (2,2) = 0 \cdot 2 + 2 \cdot 2 = 4.$$

$$S_{22} = (0,2) \cdot (1,1) = 0 + 2 = 2.$$

$$S_{23} = (0,2) \cdot (2,1) = 0 + 2 = 2.$$

Row 2: $(4, 2, 2)$.

Row 3: $Q_3 = (2,2)$.

$$S_{31} = (2,2) \cdot (2,2) = 4 + 4 = 8.$$

$$S_{32} = (2,2) \cdot (1,1) = 2 + 2 = 4.$$

$$S_{33} = (2,2) \cdot (2,1) = 4 + 2 = 6.$$

Row 3: $(8, 4, 6)$.

Thus

$$S = \begin{pmatrix} 12 & 6 & 9 \\ 4 & 2 & 2 \\ 8 & 4 & 6 \end{pmatrix}.$$

### Step 5: scale scores by $\sqrt{d_{\text{head}}}$

Here $d_{\text{head}} = 2$. Thus $\sqrt{2} = 1.4142135623730951$ (we'll keep 8–9 digits for clarity).

Divide every entry by $\sqrt{2}$:

First row:

$$\tilde{S}_{11} = \frac{12}{\sqrt{2}} \approx 8.48528137, \quad \tilde{S}_{12} = \frac{6}{\sqrt{2}} \approx 4.24264069, \quad \tilde{S}_{13} = \frac{9}{\sqrt{2}} \approx 6.36396103.$$

Second row:

$$\tilde{S}_{21} = \frac{4}{\sqrt{2}} \approx 2.82842712, \quad \tilde{S}_{22} = \frac{2}{\sqrt{2}} \approx 1.41421356, \quad \tilde{S}_{23} = \frac{2}{\sqrt{2}} \approx 1.41421356.$$

Third row:

$$\tilde{S}_{31} = \frac{8}{\sqrt{2}} \approx 5.65685425, \quad \tilde{S}_{32} = \frac{4}{\sqrt{2}} \approx 2.82842712, \quad \tilde{S}_{33} = \frac{6}{\sqrt{2}} \approx 4.24264069.$$

### Step 6: softmax (row-wise) — compute probabilities

We compute softmax on each row. For numerical stability subtract the row max first.

**Row 1:** values (8.48528137, 4.24264069, 6.36396103). Max is 8.48528137. Subtract:

$$(0,\ -4.24264068,\ -2.12132034).$$

Exponentiate:

$$e^{0} = 1, \quad e^{-4.24264068} \approx 0.014332, \quad e^{-2.12132034} \approx 0.119019.$$

Sum: $1 + 0.014332 + 0.119019 \approx 1.133351$.

Softmax row 1:

$$A_1 = \Big(\frac{1}{1.133351},\ \frac{0.014332}{1.133351},\ \frac{0.119019}{1.133351}\Big) \approx (0.88164541,\ 0.01266889,\ 0.10568570).$$

**Row 2:** values (2.82842712, 1.41421356, 1.41421356). Max is 2.82842712. Subtract:

$$(0,\ -1.41421356,\ -1.41421356).$$

Exponentiate:

$$e^{0} = 1, \quad e^{-1.41421356} \approx 0.243144, \quad e^{-1.41421356} \approx 0.243144.$$

Sum: $1 + 0.243144 + 0.243144 \approx 1.486288$.

Softmax row 2:

$$A_2 \approx (0.6728418,\ 0.1635791,\ 0.1635791).$$

**Row 3:** values (5.65685425, 2.82842712, 4.24264069). Max is 5.65685425. Subtract:

$$(0,\ -2.82842712,\ -1.41421356).$$

Exponentiate:

$$e^{0} = 1, \quad e^{-2.82842712} \approx 0.059015, \quad e^{-1.41421356} \approx 0.243144.$$

Sum: $1 + 0.059015 + 0.243144 \approx 1.302159$.

Softmax row 3:

$$A_3 \approx (0.76791794,\ 0.04538836,\ 0.18669370).$$

So the attention matrix $A$ (rows sum to 1) is approximately:

$$A \approx \begin{pmatrix} 0.88164541 & 0.01266889 & 0.10568570 \\ 0.67284180 & 0.16357910 & 0.16357910 \\ 0.76791794 & 0.04538836 & 0.18669370 \end{pmatrix}.$$

### Step 7: final head output $AV$

Multiply $A$ (3x3) by $V$ (3x2) to get output (3x2).

Compute row 1:

$$\text{out}_1 = 0.88164541 \cdot V_1 + 0.01266889 \cdot V_2 + 0.10568570 \cdot V_3.$$

Recall $V_1 = (2, 2)$, $V_2 = (1, 1)$, $V_3 = (1, 2)$.

Compute first component:

$$\text{out}_{1,1} = 0.88164541{\cdot}2 + 0.01266889{\cdot}1 + 0.10568570{\cdot}1 = 1.76329082 + 0.01266889 + 0.10568570 = 1.88164541.$$

Second component:

$$\text{out}_{1,2} = 0.88164541{\cdot}2 + 0.01266889{\cdot}1 + 0.10568570{\cdot}2 = 1.76329082 + 0.01266889 + 0.21137140 = 1.98733111.$$

Row 1 output: (1.88164541, 1.98733111).

Row 2:

$$\text{out}_2 = 0.67284180 \cdot V_1 + 0.16357910 \cdot V_2 + 0.16357910 \cdot V_3.$$

First component:

$$= 0.67284180 \cdot 2 + 0.16357910 \cdot 1 + 0.16357910 \cdot 1 = 1.3456836 + 0.1635791 + 0.1635791 = 1.6728418.$$

Second component:

$$= 0.67284180 \cdot 2 + 0.16357910 \cdot 1 + 0.16357910 \cdot 2 = 1.3456836 + 0.1635791 + 0.3271582 = 1.8364209.$$

Row 2 output: (1.6728418, 1.8364209).

Row 3:

$$\text{out}_3 = 0.76791794 \cdot V_1 + 0.04538836 \cdot V_2 + 0.18669370 \cdot V_3.$$

First component:

$$= 0.76791794 \cdot 2 + 0.04538836 \cdot 1 + 0.18669370 \cdot 1 = 1.53583588 + 0.04538836 + 0.18669370 = 1.76791794.$$

Second component:

$$= 0.76791794 \cdot 2 + 0.04538836 \cdot 1 + 0.18669370 \cdot 2 = 1.53583588 + 0.04538836 + 0.37338740 = 1.95461164.$$

Row 3 output: (1.76791794, 1.95461164).

So final head output:

$$\text{Attention}(Q, K, V) \approx \begin{pmatrix} 1.88164541 & 1.98733111 \\ 1.67284180 & 1.83642090 \\ 1.76791794 & 1.95461164 \end{pmatrix}.$$

**Summary of numeric example:** Every intermediate matrix ( $Q, K, V, S, \tilde{S}, A$ ) is shown and the arithmetic is fully expanded. This demonstrates exactly how to compute attention for small toy values and can be used to unit-test an implementation.

## 10 Backprop intuition (concise)

Attention is differentiable everywhere (softmax is smooth). Gradients flow through:

$$\frac{\partial \mathcal{L}}{\partial W_Q}, \quad \frac{\partial \mathcal{L}}{\partial W_K}, \quad \frac{\partial \mathcal{L}}{\partial W_V},$$

via chain-rule through $A = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}} + M)$ and $AV$. Pay attention that softmax Jacobian is dense per row: efficient implementations avoid naive Jacobian and instead use matrix-vector identities.

## 11 Implementation notes and gotchas

- **Compute multi-head efficiently**: compute $Q, K, V$ with single linear layers producing shape $[N, H, d_{\text{head}}]$ by a single matmul and reshape, rather than separate loops for heads.
- **Masking and fp16**: use a large finite negative number (e.g., $-1e9$) for mask to avoid producing NaN in fp16 when exponentiating $-\infty$.
- **Stable softmax**: subtract max per row before exponentiating.

- **Tied embeddings**: tying input embeddings $E$ and output linear projection (transposed) saves memory and sometimes improves generalization.
- **Pre-norm recommended**: for deep stacks (hundreds of layers), pre-norm (LayerNorm before sublayer) is more stable.
- **KV cache memory**: each layer caches $K, V \in \mathbb{R}^{T \times d_{\text{head}}}$ per head. For long contexts this can be memory-heavy.

## 12 PyTorch-style pseudocode (concise, clear shapes)

```
# INPUT:
# x: Tensor of shape (batch, seq_len, d)
# mask: Tensor of shape (batch, seq_len, seq_len) with 0 where allowed, -1e9 where blocked

# single linear to get QKV, then reshape
qkv = linear_qkv(x)  # (batch, seq_len, 3*d)
qkv = qkv.view(batch, seq_len, 3, H, d_head)
qkv = qkv.permute(2, 0, 3, 1, 4)  # (3, batch, H, seq_len, d_head)
Q, K, V = qkv[0], qkv[1], qkv[2]  # each (batch, H, seq_len, d_head)

# compute scores
scores = torch.matmul(Q, K.transpose(-2,-1))  # (batch, H, seq_len, seq_len)
scores = scores / math.sqrt(d_head)
scores = scores + mask  # broadcasting mask
attn = torch.softmax(scores, dim=-1)  # along last dim

# weighted sum
out = torch.matmul(attn, V)  # (batch, H, seq_len, d_head)
out = out.permute(0, 2, 1, 3).contiguous().view(batch, seq_len, d)
out = linear_out(out)  # (batch, seq_len, d)
```

## 13 Decoding and sampling strategies (short)

- **Greedy**: pick argmax each step.
- **Temperature**: divide logits by temperature $\tau$ before softmax.
- **Top-k**: restrict to top-k logits.
- **Top-p (nucleus)**: restrict to smallest set whose cumulative prob $\geq p$.

## 14 Putting it all together: the full decoder-only block (diagram)

## 15 Advanced topics (brief pointers)

- **Sparse attention** (local/blockwise) for very long sequences.
- **Relative positional encodings** for better generalization to unseen lengths.
- **Mixture of Experts** to scale model capacity efficiently.
- **Efficient kernels / flash attention** to reduce memory bandwidth and runtime.

## Appendix A — Summary cheat-sheet

- $Q = XW_Q$, $K = XW_K$, $V = XW_V$.
- $S = QK^\top$, $\tilde{S} = S/\sqrt{d_k} + M$.
- $A = \text{softmax}(\tilde{S})$ row-wise.
- $\text{head} = AV$, $\text{MultiHead} = \text{Concat}(\text{heads})W_O$.
- FFN: $W_2\,\sigma(W_1 x + b_1) + b_2$.

**If you want:** I can now

1. Expand the PyTorch pseudocode into a full runnable script (training loop, loss, optimizer, data batching).
2. Produce a version of this manual optimized for printing (larger diagrams, two-column layout).
3. Add a section "Debugging checklist" with unit tests and tiny code assertions (e.g., check shapes, row sums of attention equal 1, masking tests).

Tell me which and I'll produce it immediately.