JpGraph Barcode Architecture

> **This document provides a comprehensive technical description of the four barcode subsystems included in JpGraph 3.1.6p.** It covers the 2D symbologies — PDF417, QR Code, and Data Matrix — as well as the suite of 1D linear barcodes. The goal is to make the internal architecture, encoding algorithms, and design decisions accessible to a reader who may not be intimately familiar with how barcodes are constructed.

Table of Contents

1. Barcode Basics

A barcode is a machine-readable encoding of data into a visual pattern. The two broad families are:

- **1D (linear) barcodes** encode data in a sequence of bars and spaces of varying widths arranged along a single line. The classic supermarket barcode (UPC/EAN) is the canonical example. Each symbology defines a mapping from characters to bar/space patterns, plus guard bars to frame the data, and often an error-detection checksum.

- **2D barcodes** encode data in a two-dimensional grid of dark and light modules (tiny squares). They can carry far more data than 1D barcodes — from a few dozen bytes (Data Matrix, QR Code) to over a kilobyte (PDF417). 2D barcodes include **error correction** that allows the data to be recovered even when parts of the symbol are damaged.

All barcode encoders in JpGraph follow the same general pipeline:

```
Input Data → Validation → Encodation → Error Correction → Symbol Layout →
Backend Rendering
```

The sections below describe each barcode family's implementation in detail.

2. Common Architecture

Every barcode module in JpGraph follows a three-stage architecture:

Stage 1 — Encoder

The encoder transforms input data (a character string or byte array) into an internal representation. This involves:

- **Validation** — verifying that the input conforms to the symbology's character set and length constraints.
- **Encodation** — converting characters into codewords or bit patterns using the symbology's encoding rules.
- **Error correction** — computing and appending redundant codewords (Reed-Solomon, convolutional codes, or simple checksums) so that a scanner can recover the original data even if parts of the symbol are damaged or obscured.

Stage 2 — Print Specification

The encoder produces an intermediate data structure — a *print specification* — that captures the symbol in a format-independent way. For 1D barcodes this is a `BarcodePrintSpec` containing an array of bar/space widths. For 2D barcodes it is a `PrintSpecification` (Data Matrix) or `QRMatrixLayout` (QR Code) holding a two-dimensional bit matrix.

This separation means the encoding logic is completely independent of how the final image is produced.

Stage 3 — Output Backend

A backend takes the print specification and renders it to a specific output format. Every barcode module supports at least two backends:

Backend	Format	Rendering Technology
IMAGE	PNG, JPEG, GIF, WBMP	PHP GD library (`imagecreatetruecolor`, `imagefilledrectangle`, etc.)
PS	PostScript	Generated PS program text — bars drawn as `moveto`/`rlineto`/`stroke`
EPS	Encapsulated PostScript	PS with `%%BoundingBox` and without `showpage`
ASCII	Text (2D only)	Character grid using `X` and `-` (or configurable glyphs)

All backends share a common base class that provides configuration methods for module width, colors, scale, rotation, quiet zones, and whether to render human-readable text beneath the bars.

A **factory class** instantiates both the encoder and the backend, so client code never needs to know which concrete classes are involved:

```
BarcodeFactory::Create(ENCODING_CODE128)   → BarcodeEncode_CODE128
BackendFactory::Create('IMAGE', $encoder)  → OutputBackend_IMAGE
```

---

3. PDF417 — 2D Stacked Barcode

Source directory: `jpgraph-3.1.6p/src/pdf417/`

PDF417 is a *stacked* 2D barcode symbology designed by Symbol Technologies in 1991. The name stands for **P**ortable **D**ata **F**ile, with each pattern encoding data in codewords of **4** bars and spaces across **17** modules. A PDF417 symbol looks like a series of narrow 1D barcodes stacked vertically — hence "stacked" rather than "matrix."

3.1 File Organization

File	Purpose
`jpgraph_pdf417.php`	Top-level entry point and the `PDF417Barcode` encoder class
`pdf417_compressors.inc.php`	The three compaction modes: Text, Numeric, and Byte
`pdf417_clusters.inc.php`	Cluster codeword pattern tables (three clusters of 929 patterns each)
`pdf417_backends.inc.php`	IMAGE and PS/EPS output backends
`pdf417_error.inc.php`	Reed-Solomon error correction codeword generator

3.2 Encoding Pipeline

```
Input string
  → PrepData()          Segment the input into Text / Numeric / Byte
chunks
  → Enc()               Compress each segment with the appropriate
compactor
  → Reed-Solomon        Compute error correction codewords
  → Row assembly        Build rows of symbols with left/right indicators
  → Backend.Stroke()    Render to image or PostScript
```

Step 1 — Data Segmentation (`PrepData`)

The encoder analyzes the input string character by character and partitions it into segments, each tagged with the most efficient compaction mode:

Mode	Tag	Used For
Text Compaction (TC)	`USE_TC`	Printable ASCII (32–126), plus CR, LF, TAB
Numeric Compaction (NC)	`USE_NC`	Runs of 13 or more consecutive digits
Byte Compaction (BC)	`USE_BC_E6` / `USE_BC_06`	Binary data outside the printable range

The segmentation logic uses a threshold: a run of digits shorter than 13 is left in Text mode because the overhead of switching to Numeric mode would negate the higher compression ratio. For short binary sequences (1–2 bytes), a *shift* to Binary is used rather than a full mode latch, avoiding the overhead of switching modes and switching back.

Step 2 — The Text Compaction State Machine

The Text Compressor in `pdf417_compressors.inc.php` implements a **state machine** with four sub-modes:

Sub-mode	Character Set
Alpha	A–Z, space
Lower	a–z, space
Mixed	0–9 and `& \r \t , : # - . $ / + % * = ^`, space
Punctuation	`;< > @ [ \ ] _ \` ~ ! \n "

The encoder starts in Alpha sub-mode. When it encounters a character that belongs to a different sub-mode, it must decide whether to *shift* (temporary — one character, then automatically return) or *latch* (permanent — stay in the new sub-mode). This decision is governed by a **state transition table** that encodes the optimal strategy for every pair of (current sub-mode, target sub-mode):

- **Single character in a different sub-mode** → prefer a shift if one exists (costs one symbol vs. a permanent latch).
- **Two or more consecutive characters in a different sub-mode** → latch to the new sub-mode (amortizes the switching cost).

The transition table also handles multi-step latches. For example, latching from Punctuation to Lower requires two transitions: first latch to Alpha (the only exit from Punctuation), then latch to Lower. The state machine computes these multi-step paths automatically.

Each pair of text sub-mode values (each in the range 0–29) is packed into a single PDF417 codeword value:

$$\text{codeword} = v_1 \times 30 + v_2$$

This encoding doubles the density compared to naïve one-character-per-codeword approaches.

Why this design is notable: The state-machine-driven text compressor is both *optimal* (every transition uses the fewest possible symbols) and *self-contained* (the transition table replaces what would otherwise be a complex nest of conditional logic). A single table lookup decides the action for any sub-mode transition, and the compressor can be extended or modified simply by editing the table. This table-driven approach was, to the author's knowledge, a novel way to implement PDF417 text compaction at the time of writing.

Step 3 — Numeric Compaction

Numeric Compaction converts runs of digits to base-900 representation. The algorithm:

1. Prepend a leading `1` to the digit string (this preserves leading zeros).
2. Convert the resulting large integer to base 900 using PHP's arbitrary-precision `bcmod` and `bcdiv` functions.
3. Each base-900 digit becomes one codeword (value 0–899).

This achieves a density of roughly 2.93 digits per codeword — nearly three times more efficient than text mode for pure numeric data. Groups of up to 44 digits are processed at a time (the maximum that fits the base-900 representation without overflow).

Step 4 — Byte Compaction

Byte Compaction converts 6-byte groups to 5 base-900 codewords using the weight table:

$$\text{value} = \sum_{i=0}^{5} b_i \times 256^{5-i}$$

followed by conversion to base 900. Remaining bytes (fewer than 6) are encoded as raw byte values, one per codeword.

Step 5 — Reed-Solomon Error Correction

The `ReedSolomon` class in `pdf417_error.inc.php` computes error correction codewords. PDF417 uses arithmetic modulo 929 (the prime nearest to 30^2). The implementation:

1. At construction time, **verifies the integrity** of the generator polynomial coefficient tables by computing test checksums — a defensive measure against data corruption.
2. Supports error levels 0–8, producing 2^{L+1} error correction codewords (2 to 512).
3. Uses polynomial long division: for each data codeword d_i, a feedback value $t_1 = (d_i + c_{k-1}) \bmod 929$ ripples through the generator polynomial coefficients.

Step 6 — Row Assembly

The final symbol is organized into rows, each containing:

```
[Start Pattern] [Left Indicator] [Data Codewords ...] [Right Indicator]
[Stop Pattern]
```

The number of rows (3–90) and columns (1–30) are configurable. The encoder pads the last row with pad codewords (value 900) to fill the grid, then back-patches the total codeword count into the first row's first data position.

Left and right row indicators encode metadata (row number, total rows, column count, error level) using a formula defined in the PDF417 specification. Each codeword is rendered using one of three **cluster patterns** (selected based on row number modulo 3), chosen from a table of 929 patterns per cluster.

PDF417 also supports a **truncated** mode, which omits the right-hand indicator and stop pattern to produce a narrower symbol at the cost of reduced scan reliability.

3.3 Output Backends

Backend	Class	Notes

Backend	Class	Notes
IMAGE	`OutputBackendPDF417_IMAGE`	Renders each row as a sequence of filled rectangles. Supports scaling, 90° rotation, and configurable module width.
PS/EPS	`OutputBackendPDF417_PS`	Generates a PostScript program. Each row is an array of `[height, x-position, width]` triples stroked as vertical lines.

Both backends use the same `BarcodePrintSpecPDF417` intermediate representation, which stores the symbol as a 2D array of `(value, cluster, length, pattern-string)` tuples — one sub-array per row.

---

4. QR Code — 2D Matrix Barcode

Source directory: `jpgraph-3.1.6p/src/QR/`

QR Code (Quick Response Code) was originally designed by Denso Wave in 1994 for tracking automotive parts. It has since become the most widely recognized 2D barcode in the world. The JpGraph QR encoder implements the full ISO/IEC 18004 specification for versions 1–40 (21×21 through 177×177 modules) with all four error correction levels.

4.1 File Organization

File	Purpose
`qrencoder.inc.php`	Main encoder and `DataStorage` input stream class
`reed-solomon.inc.php`	Galois field arithmetic and Reed-Solomon error correction
`qrlayout.inc.php`	Matrix construction, bit placement, format/version info
`qrmask.inc.php`	8 mask patterns and penalty scoring for mask selection
`qrcapacity.inc.php`	Specification tables: capacities, block structures, alignment positions, BCH codes
`backend.inc.php`	IMAGE, PS/EPS, and ASCII output backends
`qr-utils.inc.php`	Bit manipulation utilities and tilde preprocessing
`qrexception.inc.php`	Localized exception handling with error-as-image rendering
`rgb_colors.inc.php`	Color name → RGB translation for the image backend
`driver.php`	Unit test / prototype driver

4.2 Encoding Pipeline

```
Input string
  → DataStorage       Wrap input as a stream with character-type awareness
  → Mode selection     Choose Numeric / Alphanumeric / Byte (automatic or
manual)
  → Bit stream         Encode data into a sequence of bits
  → Pad & split        Pad to byte boundary, split into codewords, fill
with 0xEC/0x11
  → Reed-Solomon       Compute error correction, interleave blocks
  → Matrix layout      Place fixed patterns (finder, timing, alignment)
  → Bit placement      Serpentine walk to place data bits
  → Mask selection     Try all 8 masks, pick the one with lowest penalty
  → Format info        Encode EC level + mask as BCH(15,5), place in
matrix
  → Backend.Stroke()   Render to image, PostScript, or ASCII
```

Step 1 — Data Encoding Modes

The encoder supports three data encoding modes, each optimized for different character sets:

Mode	Character Set	Encoding Efficiency
Numeric	Digits 0–9	3 digits → 10 bits (3.33 bits/digit)
Alphanumeric	0–9, A–Z, space, `$ % * + - . / :`	2 characters → 11 bits (5.5 bits/char)
Byte	Any 8-bit value	1 byte → 8 bits

Each encoded segment begins with a 4-bit **mode indicator** and a **character count** field whose length varies by QR version:

Version Range	Numeric Count Bits	Alphanumeric Count Bits	Byte Count Bits
1–9	10	9	8
10–26	12	11	16
27–40	14	13	16

In **automatic mode**, the encoder determines the initial encoding from the input data and dynamically switches modes when it detects a run of characters that would be more efficiently encoded in a different mode. For example, if 13 or more consecutive digits appear within alphanumeric data, the encoder latches to Numeric mode for that segment.

Step 2 — Padding and Codeword Formation

After the data bit stream is complete:

1. A 4-bit terminator (`0000`) is appended.
2. Zero bits pad the stream to a byte boundary.
3. The bit stream is split into 8-bit codewords.

4. Alternating pad bytes `0xEC` and `0x11` fill the remaining capacity (a specification requirement that ensures the symbol has a balance of dark and light modules).

Step 3 — Reed-Solomon Error Correction over GF(2^8)

The Reed-Solomon implementation is a key piece of the QR encoder. It operates over the **Galois field GF(2^8)** with primitive polynomial:

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \quad (\texttt{0x11d})$$

Galois Field Arithmetic

The `Galois` class precomputes **logarithm and antilogarithm (exponential) tables** at construction time. These tables convert multiplication in GF(2^8) — which would otherwise require polynomial multiplication and reduction — into a simple table lookup:

$$a \cdot b = \alpha^{(\log_\alpha a + \log_\alpha b) \bmod 255}$$

Addition in GF(2^8) is XOR. The antilog table is computed by iterative doubling with XOR reduction whenever the value exceeds 255:

```
α⁰ = 1
αⁱ = 2 · αⁱ⁻¹     (if < 256)
αⁱ = 2 · αⁱ⁻¹ ⊕ 0x11d    (if ≥ 256)
```

Generator Polynomial

The generator polynomial for N error correction codewords is:

$$G(x) = \prod_{i=0}^{N-1} (x - \alpha^i)$$

The implementation computes this incrementally: starting with $G(x) = 1$, it multiplies by each linear factor $(x - \alpha^i)$ using the recursive coefficient update:

$$q_n = q_{n-1}, \quad q_j = q_{j-1} \oplus (\alpha^i \cdot q_j), \quad q_0 = \alpha^i \cdot q_0$$

Systematic Encoding

Error correction codewords are computed by polynomial long division (equivalent to an LFSR-based encoder). For each data codeword d_i:

$$k = d_i \oplus \text{error}[N-1]$$

$$\text{error}[j] = \text{error}[j+1] \oplus (k \cdot G_{N-j-1})$$

The remainder after processing all data codewords gives the N error correction codewords.

Block Structure and Interleaving

For larger QR versions, data is split into multiple blocks (up to 67 blocks for version 40-L). The specification defines one or two block types per version/EC-level combination. The encoder:

1. Distributes data codewords across blocks according to the block structure table.
2. Computes Reed-Solomon error correction codewords independently for each block.
3. **Interleaves** the final sequence: data codewords from all blocks in round-robin order, then error correction codewords in round-robin order.

This interleaving ensures that a localized burst of damage affects different blocks rather than destroying one block entirely.

Step 4 — Matrix Layout

The `QRMatrixLayout` class constructs the complete QR symbol matrix. The matrix dimension is $4v + 17$ where v is the version number (21×21 for version 1, up to 177×177 for version 40).

Fixed patterns are placed first, before any data:

Pattern	Location	Purpose
Finder patterns (3)	Top-left, top-right, bottom-left corners	7×7 dark-bordered squares enabling rapid symbol detection and orientation
Quiet zone separators	1-module white border around each finder	Ensures finders are visually distinct
Timing patterns	Row 6 and column 6	Alternating dark/light modules that help scanners determine module grid
Alignment patterns	Version-specific grid positions	5×5 squares that correct for perspective distortion (version 2+)
Format information	Adjacent to top-left finder	15-bit BCH(15,5) code encoding EC level and mask pattern
Version information	Near bottom-left and top-right (version 7+)	18-bit BCH(18,6) code encoding the version number

Step 5 — Data Bit Placement

Data bits are placed into the matrix in a specific serpentine pattern:

1. Start at the **bottom-right corner** of the matrix.
2. Move in **two-column-wide paths** — right column first, then left column.
3. Alternate direction: **upward** on the first path, **downward** on the next, etc.
4. **Skip column 6** entirely (it contains the vertical timing pattern).
5. Skip any module already occupied by a fixed pattern (finder, alignment, timing, format, version).

This placement order was carefully designed by the QR specification authors to distribute data evenly across the symbol.

Step 6 — Mask Selection and Application

Raw data placement can produce patterns that confuse scanners (for example, accidental finder-pattern look-alikes, or large areas of uniform color). To mitigate this, the QR specification defines **8 mask patterns**, each a mathematical function that determines which data modules to toggle:

Mask	Condition (toggle if true)
0	$(i+j) \bmod 2 = 0$
1	$i \bmod 2 = 0$
2	$j \bmod 3 = 0$
3	$(i+j) \bmod 3 = 0$
4	$(\lfloor i/2 \rfloor + \lfloor j/3 \rfloor) \bmod 2 = 0$
5	$(ij) \bmod 2 + (ij) \bmod 3 = 0$
6	$((ij) \bmod 2 + (ij) \bmod 3) \bmod 2 = 0$
7	$((ij) \bmod 3 + (i+j) \bmod 2) \bmod 2 = 0$

where i = row and j = column.

The encoder **tries all 8 masks**, evaluates each against a **penalty scoring system**, and selects the mask with the lowest total score. The penalty criteria are:

Feature	Weight	What It Penalizes
Adjacent same-color runs (≥ 5)	$N_1 = 3$	Long horizontal or vertical stripes
Same-color rectangular blocks	$N_2 = 3$	Large uniform areas
1:1:3:1:1 dark patterns	$N_3 = 40$	Patterns that look like finder markers
Dark/light proportion deviation	$N_4 = 10$	Imbalanced overall color ratio

The `QRMask` class is implemented as a singleton to avoid recreating the evaluation logic for each encoding operation.

Step 7 — Format and Version Information

After mask selection, the encoder places two pieces of metadata into the matrix:

- **Format information** (15 bits): Encodes the error correction level (2 bits) and mask pattern index (3 bits), protected by a BCH(15,5) error-correcting code and XORed with the mask `101010000010010` to avoid all-zero patterns. This is placed in two copies for redundancy — one adjacent to the top-left finder, the other split between the bottom-left and top-right finders.

- **Version information** (18 bits, version 7+ only): Encodes the version number (6 bits) protected by a BCH(18,6) code. Placed in two copies near the bottom-left and top-right corners.

Both BCH code tables are precomputed and stored in `QRCapacity` for fast lookup.

4.3 Defensive Design

The QR encoder is notably defensive in its implementation:

- **Integrity verification**: The `ReedSolomon` class verifies its generator polynomial coefficient tables at construction time by computing test checksums.
- **Capacity tables**: All 160 entries (40 versions × 4 EC levels) for data capacity and all 160 entries for block structure are stored as explicit arrays in `QRCapacity`, rather than computed — eliminating algorithmic errors and enabling fast lookup.
- **Alignment pattern validation**: After computing alignment pattern positions, the code performs a sanity check comparing the count against the expected number.
- **Error-as-image rendering**: If any error occurs during encoding (invalid data, version overflow, etc.), the exception handler renders the error message as a GD image — ensuring that calling code always receives an image, even on failure.
- **Bit count assertions**: Before bit placement, the encoder verifies that the total bit count exactly matches the expected capacity. After placement, the `flatten()` method verifies that no uninitialized modules remain.

4.4 Output Backends

Backend	Class	Notes
IMAGE	`QRCodeBackend_IMAGE`	Renders each module as a filled GD rectangle. Configurable dark/light/background colors.
PS	`QRCodeBackend_PS`	Generates PostScript. Each row emits an array of x-positions for dark modules.
EPS	`QRCodeBackend_PS` (with `SetEPS`)	Same as PS but with `%%BoundingBox` and without `showpage`.
ASCII	`QRCodeBackend_ASCII`	Text rendering using configurable dark/light characters.

All backends are instantiated through `QRCodeBackendFactory::Create()`.

---

5. Data Matrix — 2D Matrix Barcode

Source directory: `jpgraph-3.1.6p/src/datamatrix/`

Data Matrix is a 2D matrix barcode symbology defined in ISO/IEC 16022. It is widely used in electronics manufacturing, pharmaceutical labeling, and postal services. JpGraph implements both the legacy **ECC 000–140** variant and the modern **ECC 200** variant — making it one of the more complete Data Matrix implementations available.

5.1 File Organization

File	Purpose
`datamatrix.inc.php`	Top-level entry point, constants, and `DatamatrixFactory`
`datamatrix-140.inc.php`	ECC 140 encoder orchestrator
`datamatrix-200.inc.php`	ECC 200 encoder orchestrator

File	Purpose
`encodation-140.inc.php`	ECC 140 encodation schemes (Base-N conversion)
`encodation-200.inc.php`	ECC 200 encodation schemes (ASCII, C40, TEXT, X12, BASE256, AUTO)
`reed-solomon.inc.php`	Galois field arithmetic and Reed-Solomon for ECC 200
`conv-140.inc.php`	Convolutional forward error correction coding for ECC 140
`crc-ccitt.inc.php`	CRC-CCITT checksum for ECC 140
`master-rnd-140.inc.php`	Master random bit stream for ECC 140 randomization
`bit-placement-bin-140.inc.php`	Bit placement for ECC 140 (binary data files)
`bit-placement-bin-200.inc.php`	Bit placement for ECC 200 (binary data files, multi-region)
`printspec.inc.php`	`PrintSpecification` — the intermediate data transfer object
`backend.inc.php`	IMAGE, PS/EPS, and ASCII output backends
`dm-utils.inc.php`	Bit manipulation and tilde preprocessing
`dmexception.inc.php`	Localized exception handling
`rgb_colors.inc.php`	Color management for the image backend
`bindata/`	Precomputed binary bit-placement maps (one file per symbol size)

5.2 ECC 200 — The Modern Standard

ECC 200 is the current Data Matrix standard (the only variant supported by GS1). It supports 30 symbol sizes — 24 square (10×10 through 144×144) and 6 rectangular (8×18 through 16×48).

Encoding Pipeline (ECC 200)

```
Input string
  → Tilde preprocessing       Expand ~dNNN, ~1 (FNC1), ~7nnnnnn (ECI), etc.
  → Encodation                Convert to codewords (auto or manual mode
selection)
  → Padding                   Fill remaining capacity with randomized pad
codewords
  → Reed-Solomon              Compute error correction, interleave blocks
  → Bit array conversion      Convert codewords to a flat bit stream
  → Bit placement             Map bits into the symbol matrix (from binary
data files)
  → Alignment patterns        Add solid/alternating edge patterns and
internal separators
  → Backend.Stroke()          Render to image, PostScript, or ASCII
```

Encodation Schemes

ECC 200 defines six encodation schemes, each optimized for different data profiles:

Scheme	Codeword	Density	Best For
ASCII	1 byte → 1 codeword; 2 digits → 1 codeword	Moderate	Mixed alphanumeric data
C40	3 characters → 2 codewords	High	Uppercase + digits + space
TEXT	3 characters → 2 codewords	High	Lowercase + digits + space
X12	3 characters → 2 codewords	High	ANSI X12 EDI subset (CR, *, >, uppercase, digits, space)
EDIFACT	4 characters → 3 codewords	Highest	EDIFACT (not implemented in this version)
BASE256	1 byte → 1 codeword (+ length header)	Low	Raw binary data

The **C40 and TEXT** encodation schemes use a shared encoder with different character-to-value mapping functions. Both pack three values into two codewords using the formula:

$$\text{codeword} = 1600 \times c_1 + 40 \times c_2 + c_3 + 1$$

Characters outside the basic set require "shift" prefixes: Shift 1 for control characters, Shift 2 for punctuation, Shift 3 for lowercase (C40) or uppercase (TEXT). Extended ASCII (> 127) requires a double shift (Shift 1 + Upper Shift code 30).

Automatic Mode Selection (ISO 16022 Annex P)

In **automatic mode**, the encoder implements the look-ahead algorithm from ISO 16022 Annex P. It maintains weighted cost counters for each encoding mode and processes characters one at a time:

Mode	Cost per basic character	Cost per extended ASCII
ASCII	1 for text, 0.5 for digits, 2 for extended	2 (with ceiling)
C40	2/3	8/3
TEXT	2/3	8/3
X12	2/3 for X12 chars, 10/3 for others	13/3
BASE256	1	1

After at least 4 characters have been analyzed, the mode with the lowest accumulated cost is selected for the next segment. Latch codewords are emitted when switching modes (e.g., 230 to latch to C40, 254 to unlatch back to ASCII).

Padding

After all data is encoded, remaining capacity is filled with randomized pad codewords. The first pad is the literal value 129. Subsequent pads are computed using the **253-state randomization algorithm** from ISO 16022 Annex H.1:

$$\text{pad} = 129 + ((149 \times (\text{position}+1)) \bmod 253 + 1)$$

with the result wrapped to the range 0–254. This randomization ensures that pad codewords do not create regular patterns that could confuse scanners.

Similarly, BASE256 data bytes are randomized using a **149-state algorithm**:

$$\text{value} = \text{value} + ((149 \times (\text{position}+1)) \bmod 255) + 1$$

Reed-Solomon Error Correction

ECC 200 uses Reed-Solomon codes over **GF(2^8)** with primitive polynomial:

$$p(x) = x^8 + x^5 + x^3 + x^2 + 1 \quad (\texttt{0x12D})$$

Note that this is a *different* primitive polynomial from the one used by QR Code (`0x11d`). The generator polynomial is constructed with roots starting at α^1 (not α^0 as in QR), using the same incremental multiplication approach.

The error correction process uses **interleaving**: data codewords are distributed across 1–10 blocks in round-robin fashion, RS error codewords are computed independently per block, and the final sequence interleaves all error codewords.

The implementation documents an interesting anomaly: the 144×144 symbol's interleaving specification in ISO 16022 appears to be incorrect for some scanner implementations. A flag (`$i144NonStandard`) provides an alternative offset to accommodate this.

Bit Placement

Rather than embedding enormous PHP arrays for the bit placement mapping (which would slow PHP's parser), the implementation stores precomputed **binary data files** in the `bindata/` directory — one file per symbol size, packed as big-endian 16-bit integers. At runtime, the file is loaded with `fread` and unpacked with `unpack('n*', ...)`. A **CRC32 checksum** verifies file integrity.

For symbols larger than 24×24, the matrix is divided into multiple **data regions** separated by alignment patterns:

Symbol Size Range	Data Regions	Region Layout
10×10 – 24×24	1	Single region
26×26 – 48×48	4	2×2 grid
52×52 – 96×96	16	4×4 grid
104×104 – 144×144	36	6×6 grid
Rectangular	1 or 2	Side by side

The bit-placement algorithm maps each module in the data region to a position in the bit stream, handling the special corner cases defined by the specification (represented as sentinel values `32767` for forced-dark and `32768` for forced-light).

5.3 ECC 000–140 — The Legacy Variant

The ECC 140 variant is the original Data Matrix standard (now deprecated). It uses fundamentally different error correction and data encoding compared to ECC 200.

Encoding Pipeline (ECC 140)

```
Input string
  → Tilde preprocessing
  → Auto-select encodation schema (Base11/27/37/41/ASCII/Byte)
  → Build master bit stream: prefix(5) + CRC(16) + length(9) + data
  → Convolutional coding (ECC 050/080/100/140)
  → Prepend ECC header bits
  → XOR with master random sequence
  → Bit placement (from binary data files)
  → Add alignment edges
  → Backend.Stroke()
```

Encodation Schemes

ECC 140 uses **base-N encoding** — a form of arithmetic compression where groups of characters are treated as digits in a base-N number system:

Schema	Base	Characters	Group Size	Bits per Group
Base 11	11	Space + digits 0–9	6 chars	21 bits
Base 27	27	Space + A–Z	5 chars	24 bits
Base 37	37	Space + A–Z + 0–9	4 chars	21 bits
Base 41	41	Space + A–Z + 0–9 + `,` `-` `.` `/`	4 chars	22 bits
ASCII	256	All 7-bit ASCII	1 char	8 bits
Byte	256	All 8-bit values	1 char	8 bits

The encoder automatically selects the smallest base that can represent all input characters. Base conversion uses polynomial evaluation:

$$\text{value} = \sum_{j=0}^{N-1} P_j \times T[\text{char}_j]$$

where P_j are the positional weights (powers of the base) and T is the character-to-value mapping.

Bit Stream Structure

The ECC 140 bit stream has a specific format:

1. **5-bit prefix** — identifies the encodation schema
2. **16-bit CRC-CCITT** — checksum computed over a prefix byte plus the data
3. **9-bit data length** — bit-reversed per specification
4. **Encoded data** — each codeword also bit-reversed

The CRC-CCITT uses polynomial $x^{16} + x^{12} + x^5 + 1$ (`0x1021`) with bit-reversed input — a requirement specific to the Data Matrix specification.

Convolutional Forward Error Correction

Unlike ECC 200 (which uses Reed-Solomon), ECC 140 uses **convolutional coding** — a bit-level forward error correction technique. The implementation provides a general (n, k, m) convolutional encoder parameterized by a generator matrix:

ECC Level	Code Rate	Parameters	Strength
ECC 050	3/4	(4, 3, 3)	Lightest protection
ECC 080	2/3	(3, 2, 11)	Moderate
ECC 100	1/2	(2, 1, 15)	Strong
ECC 140	1/4	(4, 1, 13)	Strongest protection (4× expansion)

The encoding algorithm:

1. Initialize k shift registers of depth m to zero.
2. For each input cycle: load k new input bits into the registers.
3. Apply the generator matrix to produce n output bits via XOR (matrix multiplication over GF(2)).
4. Shift all registers.
5. After all data, flush the registers with $k \times m$ zero bits.

Each ECC level is a subclass of the abstract `ConvolutionCoding` base class, parameterized only by its generator matrix. The generator polynomials are taken directly from ISO 16022.

Master Random Sequence

After convolutional coding, the entire bit stream is XORed with a **fixed pseudo-random master sequence** (defined in ISO 16022 Annex C). This 277-byte table is stored in the `MasterRandom` class with CRC32 verification. The randomization ensures that the symbol has a balanced ratio of dark and light modules, which aids scanner detection.

ECC 200 does not use this step — it achieves randomization through the pad and BASE256 randomization algorithms instead.

5.4 Alignment Patterns

Both ECC 140 and ECC 200 symbols have a characteristic **L-shaped border** formed by:

- **Left edge**: solid line (all dark modules)
- **Bottom edge**: solid line (all dark modules)

- **Right edge**: alternating dark/light modules (clock track)
- **Top edge**: alternating dark/light modules (clock track)

For ECC 200 symbols with multiple data regions, internal alignment patterns (solid and alternating lines) separate the regions.

5.5 Output Backends

Backend	Class	Notes
IMAGE	`BackendMatrix_IMAGE`	Renders modules as filled GD rectangles. Supports color inversion.
PS	`BackendMatrix_PS`	Generates PostScript with metadata comments.
EPS	`BackendMatrix_PS` (with `SetEPS`)	EPS variant with bounding box.
ASCII	`BackendMatrix_ASCII`	Text rendering with configurable glyphs and module scaling.

All backends are instantiated through `DatamatrixBackendFactory::Create()`.

---

6. 1D Linear Barcodes

Source files: `jpgraph-3.1.6p/src/jpgraph_barcode.php` (core library) and `jpgraph-3.1.6p/src/barcode/mkbarcode.php` (command-line tool)

The 1D barcode module supports 13 symbologies (with 2 more stubbed for future implementation). All share a common class hierarchy rooted in the abstract `BarcodeEncode` base class.

6.1 Supported Symbologies

Symbology	Class	Character Set	Checksum	Status
EAN-13	`BarcodeEncode_EAN13`	12 digits	Weighted mod-10 (auto)	✅
EAN-8	`BarcodeEncode_EAN8`	7 digits	Weighted mod-10 (auto)	✅
UPC-A	`BarcodeEncode_UPCA`	11 digits	Weighted mod-10 (auto)	✅
UPC-E	`BarcodeEncode_UPCE`	11 digits (zero-suppressed)	Encoded in parity	✅
EAN-128 / GS1-128	`BarcodeEncode_EAN128`	Full ASCII + FNC1–4	Weighted mod-103 (auto)	✅
Code 128	`BarcodeEncode_CODE128`	Full ASCII (0–127) + FNC1–4	Weighted mod-103 (auto)	✅

Symbology	Class	Character Set	Checksum	Status
Code 39	`BarcodeEncode_CODE39`	`0-9 A-Z - . SPACE $ / + %`	Sum mod-43 (optional)	✅
Code 39 Extended	`BarcodeEncode_CODE39`	Full ASCII 0–127	Sum mod-43 (optional)	✅
Bookland (ISBN)	`BarcodeEncode_BOOKLAND`	`978` + 9 digits	EAN-13 checksum	✅
Industrial 2-of-5	`BarcodeEncode_CODE25`	Digits only	Luhn-like mod-10 (optional)	✅
Interleaved 2-of-5	`BarcodeEncode_CODEI25`	Digits only (even count)	Luhn-like mod-10 (optional)	✅
Codabar	`BarcodeEncode_CODABAR`	`0-9 - $ : / . + A B C D`	None	✅
Code 11 (USD-8)	`BarcodeEncode_CODE11`	`0-9 -`	Double C+K mod-11 (always)	✅
Code 93	`BarcodeEncode_CODE93`	—	—	❌ Stub
POSTNET	`BarcodeEncode_POSTNET`	—	—	❌ Stub

6.2 Encoding Principles

Each 1D barcode symbology defines a fixed set of bar/space patterns. The encoder looks up each input character in a symbol table and produces a **run-length encoded** string. For example, the EAN-13 pattern for the digit `0` in odd parity is `'3211'`, meaning: 3-module-wide space, 2-module bar, 1-module space, 1-module bar.

All encoders produce a `BarcodePrintSpec` — an array of tuples `(character, parity, hide_flag, pattern)` plus metadata (guard patterns, margins, human-readable text positioning).

6.3 Detailed Symbology Descriptions

EAN-13 and EAN-8 (European Article Number)

EAN is the global standard for product identification. An EAN-13 barcode encodes 13 digits (12 data + 1 checksum):

Structure:

```
[Left Guard: 111] [6 left-hand digits] [Center Guard: 11111] [6 right-hand
digits] [Right Guard: 111]
```

The first (leftmost) digit — the *number system* — is not directly encoded as bars. Instead, it determines the **parity pattern** of the 6 left-hand digits. Each left-hand digit can be encoded in "odd" (L-code) or "even" (G-code) parity, and the specific sequence of odd/even choices encodes the number system digit implicitly. Right-hand digits always use a fixed encoding (R-code).

The checksum is computed as:

$$\text{check} = (10 - (\sum_{\text{odd}} d_i \times 3 + \sum_{\text{even}} d_i \times 1) \bmod 10) \bmod 10$$

EAN-8 is the compact variant: 4 left + 4 right digits, all using odd parity (no implicit number system encoding).

UPC-A and UPC-E

UPC-A is encoded as a special case of EAN-13 with an implicit leading `0`. The number system digit and checksum digit are displayed outside the main bar pattern (in smaller text to the left and right).

UPC-E is a *zero-suppressed* version of UPC-A, designed for small packages. It reduces a 12-digit UPC-A to 6 data bars by exploiting the manufacturer/item number structure:

Compression Type	Condition	Items Possible
Type 0	Manufacturer ends in `000`, `100`, or `200`	Up to 1000
Type 1	Manufacturer ends in `X0` (X ≥ 3)	Up to 100
Type 2	Manufacturer ends in `Y0` (Y ≥ 1)	Up to 10
Type 3	All other manufacturers	5 items (5–9)

The checksum is **encoded in the parity pattern** of the 6 data symbols rather than as a separate bar — an elegant technique where one of 10 parity sequences is chosen based on the checksum value.

Code 128

Code 128 is a high-density symbology supporting the full ASCII character set. It defines three **character sets**:

Set	Encodes	Use Case
A	Control characters (0–31) + uppercase + digits	Uppercase-heavy data
B	Uppercase + lowercase + digits	Mixed-case data
C	Digit pairs (00–99) → 1 symbol each	Dense numeric sequences

The encoder performs **automatic character set optimization**:

1. If data starts with 4+ digits, begin in Set C for double-density digits.
2. Scan for control characters (→ Set A) or lowercase (→ Set B).
3. During encoding, switch to Set C whenever 4+ consecutive digits are detected.

4. For a single character requiring a different set, use SHIFT (temporary) rather than a full CODE switch (permanent).

The checksum is positionally weighted:

$$\text{check} = (\text{start_code} + \sum_{i=1}^{n} i \times \text{symbol}_i) \bmod 103$$

EAN-128 / GS1-128

EAN-128 extends Code 128 with **Application Identifiers (AIs)** — standardized data field prefixes used in supply chain management. The encoder:

1. Automatically prepends FNC1 if not already present (identifies the symbol as GS1-128).
2. Validates each AI against a table of known identifiers, checking data type (numeric vs. alphanumeric) and maximum field length.
3. Delegates bar-pattern encoding to the Code 128 engine.

The implementation supports over 40 AI prefixes, covering SSCCs, GTINs, batch/lot numbers, dates, weights, location codes, and more.

Code 39

Code 39 encodes each character as 9 elements (5 bars + 4 spaces), with 2 of the 9 being wide and 7 narrow — hence sometimes called "Code 3 of 9." The asterisk (`*`) serves as the start/stop character.

Extended Code 39 encodes the full ASCII character set (0–127) by prefixing each non-native character with a shift character. For example, lowercase `a` is encoded as the two-symbol sequence `+A`.

Industrial and Interleaved 2-of-5

Industrial 2-of-5 encodes digits only. Each digit is 5 elements (bars only), with 2 wide and 3 narrow. Spaces between characters are always narrow.

Interleaved 2-of-5 achieves higher density by encoding pairs of digits simultaneously — one digit in the bar widths, the other in the space widths. The 5-element bar pattern of digit d_1 and the 5-element space pattern of digit d_2 are interleaved into a 10-element sequence:

```
bar₁ space₁ bar₂ space₂ bar₃ space₃ bar₄ space₄ bar₅ space₅
```

This doubles the density compared to Industrial 2-of-5.

Codabar

Codabar encodes digits and six special characters (`- $ : / . +`) using 7-element patterns (4 bars + 3 spaces). Four distinct start/stop characters (A, B, C, D) allow multiple Codabar symbols to be concatenated without ambiguity. It is commonly used in blood banks and library systems.

Code 11 (USD-8)

Code 11 encodes digits and the dash character using 5-element patterns. It is notable for its **double checksum** computation:

1. **C checksum**: Weighted sum using weights cycling 1–10 (right to left), mod 11.
2. **K checksum**: Weighted sum of data + C checksum using weights cycling 1–9, mod 11.

Both check digits are always appended.

6.4 Output Backends

Backend	Class	Notes
IMAGE	`OutputBackend_IMAGE`	Renders bars as filled GD rectangles. Guard bars extend full height; data bars are shorter with human-readable text below. Supports scaling, rotation, and debug overlays.
PS/EPS	`OutputBackend_PS`	Generates PostScript. Bars are `moveto`/`rlineto`/`stroke` commands. Text is centered using PS `stringwidth`.

The `BarcodePrintSpec` intermediate format stores each character as a tuple of `(encoded-char, parity, hide-flag, run-length-pattern)`, making it straightforward for any backend to iterate through the bars.

6.5 Command-Line Tool

The file `barcode/mkbarcode.php` provides a command-line interface for generating barcode images:

```
php mkbarcode.php -b CODE128 -m 2 -y 80 -o 0 -f barcode.png "Hello World"
```

Flags control the symbology (`-b`), module width (`-m`), height (`-y`), output format (`-o`: image/PS/EPS), output file (`-f`), rotation (`-r`), and human-readable text suppression (`-x`).

---

7. Cross-Reference: The QR Specification Error

During development of the QR Code module, the author (Johan Persson) discovered **minor errors in the official QR barcode specification** (ISO/IEC 18004). These findings are documented in the `QR-paper/` directory:

- **Paper**: *"A Note on Minor Errors in the QR Standard"* — a published note documenting the discrepancies.
- **Debug log**: `qrlog.txt` — the detailed encoder debug output used to verify the implementation against the specification, showing the complete encoding process for the test string `"01234567"` (QR Version 1-M), including codeword computation, RS error correction blocks, interleaved sequences, mask evaluation scores, and the final 21×21 matrix layout.

The debug output reproduced in `qrlog.txt` illustrates the mask selection process that uncovered the specification errors. All 8 mask patterns were evaluated with their penalty scores (ranging from 1022 to

1179), and mask 7 was selected as optimal with the lowest score of 1022. The ability to produce such detailed diagnostic output was instrumental in identifying the specification discrepancies.

The Data Matrix implementation contains a related note: the 144×144 symbol's interleaving specification in ISO 16022 also appears to be incorrect for some scanner implementations, and the code includes a compatibility flag (`$i144NonStandard`) to handle the discrepancy.

This document describes the barcode implementations included in JpGraph 3.1.6p, the final Professional edition by Johan Persson (January 2010).