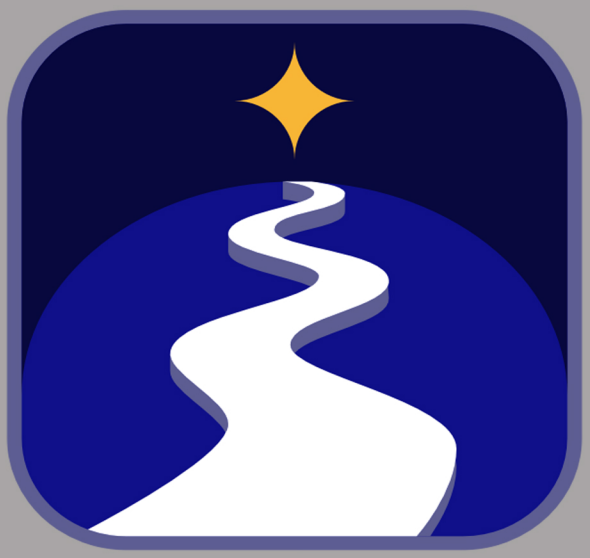

EduBASIC

User Guide v0.19

EduBASIC User Guide

EduBASIC (*pronounced **EE-doo** or **ED-yoo** BASIC*) is a modern BASIC dialect with retro roots. **EduBASIC** is designed for students to learn programming skills and explore the arts, natural sciences, and mathematics. Unlike professional programming languages, which are quite flexible and capable of implementing any sort of software, EduBASIC never leaves its walled garden. But within, we can easily create simple multi-media demos, simulations, proofs, MIDI music, and games. EduBASIC aims to make it as easy as possible to create simple software and implement algorithms.

EduBASIC includes a rich collection of program statements (74 different commands that cover variable manipulation, control flow, error handling, console I/O, graphics, and music). EduBASIC also provides a rich set of arithmetic, logic, array, and string operators (78 in total).

Table of Contents

Part I
Core EduBASIC Language 1

Part I

Core EduBASIC Language

Data Types, Values, and Variables

This chapter explains how EduBASIC represents **scalar** data in programs: the four built-in scalar types, how they **coerce** in expressions, how values are written as **literals** in source, and how **variable declaration** works through naming, sigils, defaults, and the `LET` / `LOCAL` assignment statements. Composite types — **arrays** and **structures** — build on these scalars; they are developed in ch-arrays and ch-structures.

Data Types

EduBASIC provides four built-in **scalar** data types:

- **Integer** (32-bit signed integer)
- **Real** (64-bit floating-point number)
- **Complex** (128-bit complex number with real and imaginary parts)
- **String** (text data)

Arrays and **structures** store composite data whose elements and members use these scalars (and, for structures, other structures or arrays). See ch-arrays and ch-structures.

Type Coercion

When numeric types are **mixed in an expression** or **assigned to a typed variable**, EduBASIC converts values automatically. The promotion hierarchy is **Integer → Real → Complex**: the result type is always the highest type involved. Literal syntax picks an initial type before any assignment coercion; see Literal Type Inference under Literals. The assignment rules in this section also apply when you store into a variable whose sigil fixes the type (see Variable Declaration).

Upcasting (Type Promotion)

Upcasting occurs automatically when numeric types are mixed in expressions. The result type is always the highest type in the hierarchy:

- **Integer + Integer** → Integer
- **Integer + Real** → Real
- **Integer + Complex** → Complex
- **Real + Real** → Real
- **Real + Complex** → Complex
- **Complex + Complex** → Complex

Examples:

```
LET a% = 5
LET b# = 3.14
LET c& = 2+3i

LET result1# = a% + b#  ' Integer + Real → Real (8.14)
LET result2& = a% + c&  ' Integer + Complex → Complex (7+3i)
LET result3& = b# + c&  ' Real + Complex → Complex (5.14+3i)
LET result4& = c& + c&  ' Complex + Complex → Complex (4+6i)
```

Arithmetic Operations:

- All arithmetic operations (+, -, *, /, ^, MOD) follow the same promotion rules
- Division (/): when both operands are integers and the quotient is a whole number, the result stays an **integer**; otherwise the result is **real** (for example `15 / 4` → `3.75`, but `8 / 4` → `4`)
- Modulo (`MOD`) works with both Integer and Real operands; **Integer MOD Integer** yields an integer, and when **either** operand is real the result is **real**
- Exponentiation (^) follows normal promotion rules

Examples:

```
LET q# = 15 / 4       ' Int/Int → Real (3.75)
LET m% = 17 MOD 5     ' Int MOD Int → Int (2)
LET m# = 17.5 MOD 5   ' Real MOD Int → Real (2.5)
LET m# = 17 MOD 5.5   ' Int MOD Real → Real (0.5)
LET m# = 17.5 MOD 5.5 ' Real MOD Real → Real (1.0)
LET p# = 2 ^ 8        ' Int ^ Int → Real (256)
```

```
LET mix# = 5 + 3.14   ' Int + Real → Real (8.14)
LET z& = 5 + 3i       ' Int + Complex → Complex (5+3i)
```

Downcasting (Type Demotion)

Downcasting occurs when assigning a value to a variable with a lower type in the hierarchy. Downcasting may result in loss of precision or information:

- **Real → Integer**: Truncates the decimal part (rounds toward zero)
- **Complex → Real**: Extracts only the real part, discards imaginary part
- **Complex → Integer**: Extracts real part and truncates to integer

Examples:

```
LET r# = 3.9
' Real → Integer: 3 (truncated, not rounded)
LET n% = r#

LET z& = 5.7+2.3i
' Complex → Real: 5.7 (imaginary part lost)
LET x# = z&
' Complex → Integer: 5 (real part truncated)
LET k% = z&

LET w& = 3.14+0i
' Complex → Integer: 3
LET j% = w&
```

Important Notes:

- Downcasting to Integer truncates (rounds toward zero), it does not round
- Downcasting from Complex to Real or Integer loses the imaginary component
- No automatic downcasting occurs in expressions, only in assignments

Assignment Coercion

When a value is stored with **LET** or **LOCAL**, it is coerced to match the **target variable's type** (the sigil on the name, or the structure/array shape of the target path):

```
LET i% = 42
LET i% = 42.7
LET i% = 42.7+0i
```

```
LET r# = 42
LET r# = 42.7
LET r# = 42.7+3i

LET z& = 42
LET z& = 42.7
LET z& = 42.7+3i
```

Summary of Type Coercion

Arithmetic Operations:

Operation	Result Type	Notes
Integer op Integer	Integer	Except division (/): whole-number quotients stay Integer; otherwise Real; MOD yields Integer
Integer op Real	Real	Integer promoted to Real
Integer op Complex	Complex	Integer promoted to Complex
Real op Real	Real	MOD yields Real
Real op Complex	Complex	Real promoted to Complex
Complex op Complex	Complex	MOD not applicable to complex numbers

Per-operator coercion for trigonometric, hyperbolic, exponential, and root operators is summarized in Numerical Operations.

Assignment Coercion:

Assignment	Result Type	Notes
Assign Real to Integer	Integer	Truncated (not rounded)
Assign Complex to Real	Real	Imaginary part lost
Assign Complex to Integer	Integer	Real part truncated, imaginary lost

Literals

A **literal** is a fixed spelling in source that denotes a value directly, never with a type sigil. Literals appear on the right-hand side of assignments, inside expressions, and in DATA files. The subsections below describe each scalar literal form; Literal Type Inference explains how syntax selects a type before Type Coercion adjusts a value to a variable's sigil on assignment. Bracket and brace literal forms for arrays and structures are introduced in ch-arrays and ch-structures.

Integer Literals

Integer literals can be written in three forms:

Decimal:

```
123
0
9001
```

Hexadecimal (prefix with `&H`):

```
&HFF
&H7F2A
&H00FF
```

Binary (prefix with `&B`):

```
&B101101
&B1101_0011
&B11111111
```

Underscores can be used in binary literals for readability.

Real Literals

Real (floating-point) numbers can be written as:

Decimal notation:

```
3.14
10.
.25
42
```

Scientific notation:

```
1E6
3.2E-4
6.022E23
1.5e+10
```

The exponent uses `E` or `e` followed by an optional sign (`+` or `-`).

A **decimal integer literal** (e.g. `42`) still tokenizes as an integer; when the context expects a **real** (for example `LET x# = 42`), it is **coerced** to real, similar to FreeBASIC/QB64-style numeric literals.

Complex Literals

Complex numbers consist of a real part and an imaginary part.

Imaginary-only (real part is zero):

```
4i
3.14i
.25i
1E-3i
```

Full complex numbers:

```
3+4i
3-4i
10.5+2.5E-3i
1E3-2i
.5+.25i
```

Notes:

- The imaginary unit can be lowercase `i` or uppercase `I`
- No spaces are required between the real and imaginary parts
- The + or - immediately before the imaginary term distinguishes it from subtraction in expressions

String Literals

String literals are enclosed in double quotes:

```
"Hello, world!"
"EduBASIC"
"Line 1\nLine 2"
```

The backslash in the third example begins an escape sequence (`\n` inserts a newline). Full escape rules, Unicode, and UTF-8 are covered in String Literals under Strings. To build a string from mixed text and expressions, use a **string template** (`` `$| … |$` ``); see String Templates. String templates are not string literals: they evaluate to a string.

Literal Type Inference

Before coercion, EduBASIC infers a literal's type from how it is written:

- Numbers without a decimal point or exponent → **Integer**
- Numbers with a decimal point or exponent → **Real**
- Numbers with an `i` or `I` imaginary unit → **Complex**

When that literal is assigned to a variable, the rules in Assignment Coercion may promote or truncate it to match the target sigil.

Examples:

```
LET a% = 42      ' Integer literal
LET b# = 42.0    ' Real literal
LET c# = 42      ' Integer literal coerced to Real
LET d& = 42+0i   ' Complex literal
LET e& = 3.14+2i ' Complex literal
```

Variable Declaration

EduBASIC does not require separate declaration statements for scalars. A name becomes a **module-level** variable the first time it is assigned with **LET**, or a **local** variable the first time it is assigned with **LOCAL** inside the current stack frame. If a name is **read** before any assignment, the runtime supplies a type-appropriate **default value** (see Default Values) without creating a stored binding until the first assignment.

The subsections below cover **naming**, **type sigils**, **defaults**, and the **LET / LOCAL** assignment statements for scalars. Arrays and structures reuse the same assignment forms with different naming rules; see ch-arrays and ch-structures.

Variable Names

Identifiers for variables must:

- Be alphanumeric (letters and numbers)
- End with the appropriate type sigil (scalar sigils below; arrays and structures use different rules)
- Start with a letter
- Be **case-insensitive** (`Count%` and `count%` are the same name)
- **Not** spell a reserved keyword (words the language uses as commands or operators; see the flat list in Appendix: Reserved Keywords)

The **type sigil** on the name declares the storage kind; see the next subsection.

Examples:

```
LET studentCount% = 10
LET roomTemperature# = 98.6
LET playerName$ = "Alice"
LET impedance& = 3+4i
```

Type Sigils

Type **sigils** are suffix characters on variable names. They are **required** for scalars and **never** appear on literals.

Data Type	Sigil	Example Variable	Size
Integer	%	`count%`	32-bit
Real	#	`value#`	64-bit
String	$	`name$`	Variable
Complex	&	`z&`	128-bit

Rules:

- Sigils are suffixes (appear at the end of the variable name)
- All scalar variables must have a sigil
- Literals never use sigils

Arrays extend scalar names with a `[]` rank suffix (for example `numbers%[]`); structure variables have **no** sigil. See ch-arrays and ch-structures.

Default Values

Before the first **LET** or **LOCAL** assignment to a name, a **read** of that name (`PRINT count%`, `IF score% > 0`, and so on) yields a default based on the type implied by the sigil:

Type	Default Value	Notes
Integer (%)	`0`	32-bit signed integer
Real (#)	`0.0`	64-bit floating-point
Complex (&)	0+0i	128-bit complex number (real and imaginary parts both 0)
String ($)	`""`	Empty string

Default values for empty arrays and empty structures are covered in ch-arrays and ch-structures.

Examples:

```
LET count% = 0
' Parse error: LET requires = and an expression
LET value#
LET name$ = ""
```

When a variable name is **read** before any assignment (`PRINT count%`, `IF score% > 0`, and so on), the runtime supplies the type-appropriate default from the table above without creating a stored binding until the first `LET` or `LOCAL` assignment.

Important Notes:

- Default values apply when a variable is first **read** before any assignment (not via bare `LET` without =)

LET and LOCAL statements

LET stores into **module-level** bindings that persist for the whole program. **LOCAL** uses the **same assignment syntax**, including compound forms (`=, -=, =, /=, ^=),` **but binds names in the** `current stack frame*` (`destroyed when that frame is popped, for example when a` SUB+ returns).

Syntax (LET, scalars):

```
LET variable = expression
LET variable += expression
LET variable -= expression
LET variable *= expression
```

```
LET variable /= expression
LET variable ^= expression
```

Syntax (`LOCAL`, scalars):

```
LOCAL variable = expression
LOCAL variable += expression
LOCAL variable -= expression
LOCAL variable *= expression
LOCAL variable /= expression
LOCAL variable ^= expression
```

The same `LET` / `LOCAL` keywords also accept array and structure target paths (`array[]`, `array[i]`, `structure.member`, and combined paths). Those forms are developed in ch-arrays and ch-structures.

Compound assignment: EduBASIC does **not** define separate assignment operators the way C does. Tokens such as `=, -=, =, /=, and ^= are` shorthand on LET or LOCAL*: the runtime reads the current value of the target, applies the ordinary arithmetic operator (, -, *, /, ^) with the right-hand expression, and stores the result back. They require an existing variable (or legal target path) compatible with that operation.

Rules for LET:

- Creates module-level variables that persist throughout the program
- Can assign to scalars (and, in later chapters, structures and arrays)
- The `LET` keyword is **required** in program source (lines in the Code editor); omitting it is a parse error
- Variables are created automatically on first assignment, no declaration needed
- Type is determined by the variable's sigil

Rules for LOCAL:

- Creates variables scoped to the **current stack frame**; they are destroyed when that frame is popped (for example when a `SUB` returns)
- Inside a `SUB`, locals belong to that procedure's frame; outside any `SUB`, locals belong to the bottom-level frame
- Locals do not collide with module-level names of the same spelling: frame bindings shadow or coexist according to normal name resolution
- Locals must still use correct type sigils
- Other statements that write through a variable target often accept `LOCAL` before the name (for example `INPUT LOCAL`, `CLAMP LOCAL`, `SWAP LOCAL`); see SUB and END SUB Statements (**Local Variables**) and the **LOCAL** entry in Statement Reference

Examples:

```
LET count% = 10
LET name$ = "Alice"
LET result# = 3.14159
LET complex& = 3+4i

' Compound assignment
LET total% = 100
LET total% += 10

SUB Demo ()
    LOCAL acc% = 0
    LOCAL acc% += 1    ' Same shorthand using LOCAL
END SUB
CALL Demo
```

SWAP Statement

SWAP exchanges the values held by two assignment targets in one step, without the temporary variable an exchange would otherwise need. It is the companion to **LET** and **LOCAL**: anything those statements can assign to, **SWAP** can exchange.

Syntax:

```
SWAP target1, target2
SWAP LOCAL target1, LOCAL target2
```

Each **target** uses the same left-hand shapes as **LET** and **LOCAL**. For scalars that is a variable name with its type sigil. You may prefix **LOCAL** on either operand independently, so `SWAP LOCAL a%, b%` exchanges a stack-frame binding with a module-level one.

Rules:

- Both operands must hold values of the **same type**; `SWAP` does **not** coerce, and a type mismatch is a runtime error.
- Each operand resolves to the same binding it would receive from **LET** or **LOCAL** (module-level vs current stack frame).
- Operands may be **any** valid `LET` / `LOCAL` target. Array-element and structure-member targets build on this base in SWAP with Arrays (under ch-arrays) and SWAP with Structures (under ch-structures).

Examples:

```
LET a% = 1
LET b% = 2
SWAP a%, b%
PRINT a%, b%    ' Prints 2 then 1

LET first$ = "before"
LET second$ = "after"
SWAP first$, second$

SUB SortPair ()
    LOCAL x% = 5
    LOCAL y% = 2
    IF x% > y% THEN
        SWAP LOCAL x%, LOCAL y%
    END IF
END SUB
```

Numerical Operations

This chapter covers numeric computation in EduBASIC: bitwise logic, arithmetic, comparisons on numeric scalars, unit conversion, transcendental operators, date and time, random numbers, and the `CLAMP` statement. String formatting with `STR` is introduced here for numeric scalars; the full `STR` / `VAL` reference is in STR, VAL, HEX, and BIN under Strings.

Boolean Values and Bitwise Operators

EduBASIC uses integers to represent boolean values:

- `0` represents **false**
- Any non-zero value (typically `1`) represents **true**

There is no separate boolean data type.

Operator	Description
`AND`	Output bit is 1 when both input bits are 1
`OR`	Output bit is 1 when at least one input bit is 1
`NOT`	Output bit is 1 when the input bit is 0
`XOR`	Output bit is 1 when the input bits are different
`NAND`	Output bit is 1 when at least one input bit is 0
`NOR`	Output bit is 1 when both input bits are 0
`XNOR`	Output bit is 1 when both input bits are the same
`IMP`	Output bit is 1 when first bit is 0 or second bit is 1

Since there is no separate boolean data type, all boolean operators are **bitwise operators** that work on integers.

Boolean nullary operators:

- `FALSE%` = `0` (always returns the same value)
- `TRUE%` = `-1` (always returns the same value)

Any non-zero value is treated as true in conditional expressions, but the canonical `TRUE%` value is `-1`. This is because `-1` in binary representation has all bits set to 1 (two's complement), which makes bitwise operations behave correctly. For example, `TRUE% AND TRUE%` yields `TRUE%` (`-1 AND -1 = -1`), while if `TRUE%` were `1`, then `1 AND 1 = 1` would only preserve the least significant bit.

Arithmetic and Comparison

These operators compute with numeric values, compare them, and convert between units.

Arithmetic Operators

Operator	Description	Example
``+	Addition	`LET totalSum% = 5  3+`
-	Subtraction	`LET difference% = 10 - 4`
*	Multiplication	`LET product% = 6 * 7`
/	Division	`LET quotient# = 15 / 4`
MOD	Modulo	`LET remainder% = 17 MOD 5`
^ or	Exponentiation	`LET powerResult# = 2 ^ 8`
!	Factorial	`LET factorialNum% = 5!`

Numeric promotion and division rules are summarized under Type Coercion in Data Types, Values, and Variables.

Comparing Numbers

Standard comparison operators work on numeric scalars:

Operator	Description	Returns TRUE% (-1) when…
=	Equal to	Both operands have the same value
<>	Not equal to	Operands have different values
<	Less than	Left operand is smaller than right operand
>	Greater than	Left operand is larger than right operand
<=	Less than or equal	Left operand is smaller or equal to right
>=	Greater than or equal	Left operand is larger or equal to right

Comparison operations return integer values: `FALSE%` (0) or `TRUE%` (-1).

Examples:

```
IF score% > 100 THEN
    PRINT "High score"
END IF
IF value# <> 0.0 THEN
    PRINT "Non-zero"
END IF
```

String, array, and structure comparisons are covered in Strings, ch-arrays, and ch-structures.

Unit Conversion Operators

EduBASIC provides postfix operators to convert between degrees and radians:

Operator	Description	Example
`DEG`	Convert degrees to radians	`LET radians# = 90 DEG`
`RAD`	Convert radians to degrees	`LET degrees# = PI# RAD`

Example usage with trigonometric operators:

```
LET angle# = 45 DEG
LET result# = SIN angle#
LET angleDegrees# = ASIN result# RAD
```

Transcendental and Mathematical Operators

This section collects unary numeric operators for exponentials, logarithms, trigonometry, hyperbolic operators, roots, rounding toward various modes, sign, and complex components. Binary arithmetic (`` ` ``, `-`, `*`, `/`, MOD, `^`) and `factorial` (`!+`) are summarized under Arithmetic Operators above.

For uniformly distributed random reals in [0, 1), see Random Number Generation (`RND#`).

Operator Type	Operand Type	Result Type	Notes
SIN, COS, TAN, ASIN, ACOS, ATAN	Integer/Real	Real	Standard trigonometric operators
SIN, COS, TAN, ASIN, ACOS, ATAN	Complex	Complex	Complex extensions of trigonometric operators
SINH, COSH, TANH, ASINH, ACOSH, ATANH	Integer/Real	Real	Standard hyperbolic operators
SINH, COSH, TANH, ASINH, ACOSH, ATANH	Complex	Complex	Complex extensions of hyperbolic operators
EXP, LOG, LOG10, LOG2	Integer/Real	Real	Standard exponential and logarithmic operators
EXP, LOG, LOG10, LOG2	Complex	Complex	Complex extensions of exponential and logarithmic operators
SQRT, CBRT	Integer/Real	Real	Standard root operators
SQRT, CBRT	Complex	Complex	Complex extensions of root operators

Operator Type	Operand Type	Result Type	Notes
ABS	Integer/Real	Real	Absolute value
ABS	Complex	Real	Magnitude (modulus)
ROUND, FLOOR, CEIL, TRUNC, EXPAND	Integer/Real	Real	Rounding and truncation operators
ROUND, FLOOR, CEIL, TRUNC, EXPAND	Complex	Error	Not applicable to complex numbers
SGN	Integer/Real	Integer	Sign operator
SGN	Complex	Error	Not applicable to complex numbers

Trigonometric Operators

	Description	Example
`SIN x#`	Sine of x (radians)	`LET result# = SIN angle#`
`COS x#`	Cosine of x (radians)	`LET result# = COS angle#`
`TAN x#`	Tangent of x (radians)	`LET result# = TAN angle#`
`ASIN x#`	Arcsine of x (returns radians)	`LET angle# = ASIN ratio#`
`ACOS x#`	Arccosine of x (returns radians)	`LET angle# = ACOS ratio#`
`ATAN x#`	Arctangent of x (returns radians)	`LET angle# = ATAN slope#`

Hyperbolic Operators

Operator	Description	Example
`SINH x#`	Hyperbolic sine of x	`LET result# = SINH value#`
`COSH x#`	Hyperbolic cosine of x	`LET result# = COSH value#`
`TANH x#`	Hyperbolic tangent of x	`LET result# = TANH value#`
`ASINH x#`	Inverse hyperbolic sine of x	`LET result# = ASINH value#`
`ACOSH x#`	Inverse hyperbolic cosine of x	`LET result# = ACOSH value#`
`ATANH x#`	Inverse hyperbolic tangent of x	`LET result# = ATANH value#`

Exponential and Logarithmic Operators

Operator	Description	Example
`EXP x#`	e raised to the power x	`LET result# = EXP power#`
`LOG x#`	Natural logarithm (base e)	`LET result# = LOG value#`
`LOG10 x#`	Common logarithm (base 10)	`LET result# = LOG10 value#`

Operator	Description	Example
`LOG2 x#`	Binary logarithm (base 2)	`LET result# = LOG2 value#`

Root Operators

Operator	Description	Example
`SQRT x#`	Square root of x	`LET result# = SQRT number#`
`CBRT x#`	Cube root of x	`LET result# = CBRT number#`

For complex operands, `SQRT` and `CBRT` return complex results (for example, `SQRT -1` yields `i`).

Absolute Value and Magnitude

Operator	Description	Example
`ABS x#`	Absolute value (real) or magnitude (complex)	`LET magnitude# = ABS (34i)+`

Rounding and Truncation Operators

Operator	Description	Example
`ROUND x#`	Round to nearest integer (ties round up)	`LET result# = ROUND 3.5` returns 4
`FLOOR x#`	Round toward negative infinity (−∞)	`LET result# = FLOOR 3.7` returns 3, `FLOOR -3.2` returns -4
`CEIL x#`	Round toward positive infinity (+∞)	`LET result# = CEIL 3.2` returns 4, `CEIL -3.7` returns -3
`TRUNC x#`	Round toward zero	`LET result# = TRUNC 3.9` returns 3, `TRUNC -3.9` returns -3
`EXPAND x#`	Round away from zero	`LET result# = EXPAND 3.1` returns 4, `EXPAND -3.1` returns -4
`INT x#`	Truncate toward zero (same as `TRUNC` for reals)	`LET dice% = INT (RND# * 6) 1+`

Sign Operator

Operator	Description	Example
`SGN x#`	Sign of x: -1, 0, or 1	`LET result% = SGN value#`

Complex Component Operators

Operator	Description	Example
`REAL expr`	Real component; Integer and Real operands promote to Complex (`imaginary = 0`)	`LET re# = REAL 42% : LET re# = REAL z&`
`IMAG expr`	Imaginary component; Integer and Real operands promote to Complex (`IMAG 5 → 0`)	`LET im# = IMAG z&`
`CONJ expr`	Complex conjugate; numeric operands promote to Complex	`LET conjugate& = CONJ z&`
`CARG expr`	Argument (phase angle) in radians; numeric operands promote to Complex	`LET phase# = CARG z&`

Mathematical Constants

- `PI#` — Returns the mathematical constant π (pi) as a real number (approximately 3.141592653589793). This nullary operator always returns the same value.
- `E#` — Returns the mathematical constant e (Euler's number) as a real number (approximately 2.718281828459045). This nullary operator always returns the same value.

```
LET circumference# = 2 * PI# * radius#
LET area# = PI# * radius# * radius#
LET exponential# = E# ^ power#
```

Date and Time

EduBASIC provides three nullary operators for date and time: `DATE$`, `TIME$`, and `NOW%`. Each evaluation returns the current value at runtime. All three use UTC.

`DATE$` - Current date as string:

```
LET today$ = DATE$
PRINT "Today is: ", today$
```

Returns the current UTC date as a string in `YYYY-MM-DD` format (ISO 8601 date format).

`TIME$` - Current time as string:

```
LET now$ = TIME$
PRINT "Time: ", now$
```

Returns the current UTC time as a string in `HH:MM:SS` format (24-hour format).

`NOW%` - Current timestamp:

```
LET timestamp% = NOW%
PRINT "Timestamp: ", timestamp%
```

Returns the current Unix timestamp (seconds since January 1, 1970, 00:00:00 UTC) as an integer. Useful for calculating time differences and storing absolute time values.

Examples:

```
' Display current date and time
PRINT "Date: ", DATE$
PRINT "Time: ", TIME$

' Calculate elapsed time
LET startTime% = NOW%
SLEEP 2000    ' Wait 2 seconds
LET endTime% = NOW%
LET elapsed% = endTime% - startTime%
PRINT "Elapsed: ", elapsed%, " seconds"
```

```
' Log with timestamp
SUB LogMessage (msg$)
    PRINT DATE$, " ", TIME$, " - ", msg$
END SUB

LogMessage "Program started"
```

```
' Check if it's a specific date
IF DATE$ = "2025-12-25" THEN
    PRINT "Merry Christmas!"
END IF
```

Random Number Generation

EduBASIC provides the `RND#` operator to generate random numbers and the `RANDOMIZE` command to seed the random number generator.

Random Number Nullary Operator:

- `RND#` - Returns a random real number in the range [0, 1). This nullary operator returns a different value on each evaluation.

Random Number Generator Seed:

The RANDOMIZE statement initializes the random number generator with a seed value.

Syntax:

```
RANDOMIZE
RANDOMIZE seedValue%
```

Rules:

- Without an argument, seeds the generator with the current Unix timestamp (NOW%)
- With an argument, seeds the generator with the specified integer value
- Using the same seed value produces the same sequence of random numbers
- Different seeds produce different random sequences

Examples:

```
' Seed with current timestamp
RANDOMIZE
RANDOMIZE 12345    ' Seed with specific value (reproducible)
RANDOMIZE NOW%    ' Explicitly seed with current timestamp
```

```
RANDOMIZE
LET randomNumber# = RND#

RANDOMIZE 12345
LET dice% = INT (RND# * 6) + 1

LET randomInRange# = RND# * 100

' Seed with current time
RANDOMIZE NOW%
LET currentTime% = NOW%
```

Formatting and Clamping

These statements turn numbers into text for display and constrain values to a range.

Formatting Numbers as Text

STR turns a numeric value into the same literal text you would type in the Code editor, for log messages and string concatenation:

```
LET num% = 42
LET numStr$ = STR num%      ' "42"
```

```
LET piStr$ = STR 3.14159    ' "3.14159"
LET complexStr$ = STR(3+4i) ' "3+4i"
LET message$ = "Result: " + STR value#
```

VAL parses a numeric literal from a string. Full rules for VAL (and STR on strings, arrays, and structures) are in STR, VAL, HEX, and BIN under Strings.

```
LET count% = VAL "42"
LET ratio# = VAL "3.14"
```

CLAMP Statement

After variables exist, **CLAMP** restricts a **scalar** integer or real variable to a closed interval in one step. The two bounds may be given in either order; the effective range runs from the smaller bound to the larger.

Syntax:

```
CLAMP variable FROM expression TO expression
CLAMP LOCAL variable FROM expression TO expression
```

Rules:

- The target must be a simple variable name (with its type sigil), not a structure member or array element.
- The variable's current value and both bound expressions must be **integer or real**. String, complex, array, and structure variables are not allowed.
- After evaluation, the variable is updated so its numeric value lies in the closed interval from the smaller bound to the larger bound.
- If the variable is an integer, the result is stored as an integer (TRUNC-style toward zero after clamping). If it is real, the result is stored as a real.
- Reads and writes use the same variable the name would refer to elsewhere (for example, a local in the current SUB when that name is local there, excluding by-reference parameter aliasing which continues to follow the existing assignment rules).

Examples:

```
LET score% = 150
CLAMP score% FROM 0 TO 100
PRINT score%          ' Prints 100

LET x# = 2.5
```

```
CLAMP x# FROM 10 TO 0 ' Same interval as FROM 0 TO 10
PRINT x#              ' Prints 10
```

Strings

You met the **String** scalar type in Data Types, Values, and Variables. This chapter covers Unicode and UTF-8, string literals and escape sequences, string templates, string operators, and scalar rules for `STR` and `VAL`.

Strings as Text Data

An EduBASIC string holds a sequence of **characters** (Unicode scalar values), not a raw byte array. Operators such as `LEN`, `LEFT`, and `MID` count and slice by **character index**; storage and file I/O use UTF-8 encoding, described in Unicode and UTF-8.

Unicode and UTF-8

Code Points and Characters

Each character in a string corresponds to one **Unicode scalar value** (a code point from `0` through `&H10FFFF`). Most everyday text lives in the **Basic Multilingual Plane** (BMP, `U0000…UFFFF`). Characters outside the BMP — including many emoji — still occupy **one** character position in EduBASIC strings and in `LEN` counts.

Isolated UTF-16 **surrogate** code unit values (`&HD800`–`&HDFFF`) are not valid scalar values; `CHR` rejects them.

UTF-8 Encoding

On disk and in the virtual file system, text is stored as **UTF-8** bytes:

- Program source on the **Code** tab
- Files opened in the Disk editor **Text** mode
- `TEXT` file records in File I/O (`READ` / `WRITE`)

UTF-8 is a variable-width encoding: ASCII letters use one byte per character; many symbols and all non-BMP characters use two to four bytes. EduBASIC strings in memory are **not** byte arrays — the runtime converts between UTF-8 (files) and scalar sequences (string values) automatically.

Character Count vs Byte Count

`LEN text$` returns the number of **characters** (scalar values), not UTF-8 bytes. A string containing one emoji has `LEN` **1** even though UTF-8 stores four bytes for that character.

For **byte** positions in a text file, use `LOC` and `SEEK` on an open `TEXT` handle (see File I/O). Do not assume `LEN` matches file byte length for non-ASCII text.

CHR, ASC, and Scalar Values

`CHR` builds a one-character string from a numeric scalar value; `ASC` reads the scalar value of the first character. Both use full code points (not UTF-16 surrogate halves). See CHR and ASC.

Unicode in Source and Programs

You can type Unicode characters directly in string literals (when your editor supports them) or build them with `CHR`:

```
LET slash$ = CHR &H2571   ' Box-drawing / (9585)
LET grin$ = CHR &H1F600     ' One emoji
LET mapLine$ = "  │  "
```

Box-drawing maps and emoji use this mix of literal glyphs and `CHR`.

String Literals

You saw basic string literals in Data Types, Values, and Variables. This section documents quoting, escapes, and round-trip rules.

Quoting and Single-Line Rules

- String literals use double quotes: `"Hello"`
- A literal cannot span source lines. An unescaped newline inside quotes is a parse error ("Unterminated string"). Use `\n` inside the literal for line breaks.
- The same escape rules apply when the tokenizer reads string tokens in expressions and in EBON data.

Escape Sequences

Inside a string literal, backslash (`\+`) starts an **escape sequence**. EduBASIC supports exactly these five:

Escape in source	Character stored	Meaning
`\n`	Line feed (U+000A)	Newline
`\t`	Tab (U+0009)	Horizontal tab
`\r`	Carriage return (U+000D)	Carriage return
`\"`	`"`	Literal double quote inside the string
`\\`	`\`	Literal backslash

Unknown escapes: If `\` is not followed by `n`, `t`, `r`, `"`, or `\`, the backslash is **discarded** and the next character is copied literally. For example, `"Unknown\xescape"` becomes `Unknownxescape` (not an error). EduBASIC does **not** support C-style `\x`, `\u`, or octal escapes in string literals; use `CHR` for code points you cannot type directly (see CHR and ASC).

Round-trip: `STR` and the Code editor re-emit `\n`, `\r`, `\t`, `\"`, and `\\` when serializing string values.

Examples:

```
LET greeting$ = "Hello, world!"
LET twoLines$ = "Line 1\nLine 2"
LET quoted$ = "Say \"Hello\""
LET path$ = "folder\\file"
```

String Templates

String templates combine quoted literal text with **`{ … }`** interpolation holes. Evaluating a string template produces an ordinary string value (usable with `LET`, `` ` ``, PRINT+, and so on).

Syntax:

```
$| "literal text" {expression} "more text" |$
$| {expression | width} |$
$| {expression | width:decimals} |$
$| {expression |< width} |$
$| {expression |> width} |$
```

Rules:

- The whole template must fit on one source line (use `\n` inside quoted chunks for line breaks).
- Fixed text appears only in normal `"…"` string chunks; holes use `{ … }`.
- `{expression}` alone inserts the value using the same display rules as `PRINT` (no padding).
- `{expression | width}` pads the displayed value into a field of `width` **display columns** (wide glyphs count as 2). Integer and real values are zero-filled on the left when right-aligned; string values use spaces.
- Bare `|` before the width uses type defaults: strings are **centered**, numbers are **right-aligned**. `|<` forces left alignment; `|>` forces right alignment.
- `{expression | width:decimals}` applies fixed decimal places on integer or real values (`width:0` is the same as `width` alone).
- When the displayed value is wider than the field, text is clipped from the far side (centered fields split the excess evenly).

Examples:

```
LET greeting$ = $| "Hello, " {name$} "!" |$
LET row$ = $| {title$ | 40} |$
LET table$ = $| {name$ |< 22} {score% | 8} {avg# | 10:2} |$
```

String Operators

EduBASIC provides several operators and forms for working with string data (concatenation, slicing, comparison, conversion, and search). For emitting strings and values to the text grid, see PRINT Statement under Console and Text I/O. Per-operator syntax lines also appear in Appendix: Operator Reference.

String Concatenation

Strings can be concatenated using the `` operator or by using commas in PRINT+ statements.

Syntax:

```
LET result$ = string1$ + string2$
LET result$ = string1$ + "literal" + string2$
```

Examples:

```
LET firstName$ = "John"
LET lastName$ = "Doe"
LET fullName$ = firstName$ + " " + lastName$
PRINT fullName$

LET greeting$ = "Hello, " + name$ + "!"
LET message$ = "Value: " + STR number%
```

String Repetition

Multiplying a string by an integer or real count repeats the string that many times. Either operand order works.

Syntax:

```
LET line$ = "-" * 40
LET line$ = 40 * "-"
LET banner$ = "Go! " * 3
```

Examples:

```
PRINT 2 * "Hello"
LET stars$ = "*" * 10
LET pad$ = " " * 4 + "text"
```

The count must be non-negative. Real counts are truncated toward zero (same rule as `LEFT` length). `0` times any string yields an empty string.

LEFT, RIGHT, and MID

EduBASIC provides string operators to extract and manipulate substrings:

`LEFT` - Extract left portion of string:

```
LET text$ = "Hello, world!"
LET firstWord$ = text$ LEFT 5
LET firstChar$ = text$ LEFT 1
```

Returns the leftmost `n` characters of the string. If `n` is greater than the string length, returns the entire string.

`RIGHT` - Extract right portion of string:

```
LET text$ = "Hello, world!"
LET lastWord$ = text$ RIGHT 6
LET lastChar$ = text$ RIGHT 1
```

Returns the rightmost n characters of the string. If n is greater than the string length, returns the entire string.

MID - Extract substring from middle (0-based, inclusive):

```
LET text$ = "Hello, world!"
LET world$ = text$ MID 7 TO 11
LET middle$ = text$ MID 2 TO 5
```

Returns a substring from index start% to index end% (both **0-based**, inclusive). If start% is beyond the string length, returns an empty string. If end% extends beyond the string, returns characters up to the end of the string.

Single character access:

```
LET text$ = "Hello"
LET char$ = text$ MID 0 TO 0
```

String Length

Use unary LEN to get the length of a string (character count). See Character Count vs Byte Count for the relationship to UTF-8 bytes.

Syntax:

```
LET length% = LEN string$
```

Example:

```
LET text$ = "Hello"
LET size% = LEN text$
```

For array element count, LEN uses the same spelling on an entire-array operand; see ch-arrays.

String Comparison

Strings can be compared using standard comparison operators (=, <>, <, >, <=, >=). Comparisons are case-sensitive and use lexicographic (alphabetical) ordering.

Examples:

```
IF name$ = "Alice" THEN
    PRINT "Found Alice"
END IF
IF text1$ <> text2$ THEN
    PRINT "Different"
END IF
```

```
IF word$ < "middle" THEN
    PRINT "Comes before 'middle'"
END IF
```

STARTSWITH and ENDSWITH

`STARTSWITH` - Check if string starts with prefix:

```
LET filename$ = "document.txt"
IF filename$ STARTSWITH "doc" THEN
    PRINT "Starts with 'doc'"
END IF
IF filename$ STARTSWITH ".txt" THEN
    PRINT "Starts with '.txt'"
END IF
```

Returns TRUE% (-1) if the string starts with the specified prefix, FALSE% (0) otherwise. Case-sensitive.

`ENDSWITH` - Check if string ends with suffix:

```
LET filename$ = "document.txt"
IF filename$ ENDSWITH ".txt" THEN
    PRINT "Ends with '.txt'"
END IF
IF filename$ ENDSWITH "doc" THEN
    PRINT "Ends with 'doc'"
END IF
```

Returns TRUE (-1) if the string ends with the specified suffix, FALSE (0) otherwise. Case-sensitive.

INSTR and REPLACE

`INSTR` - Find substring position (0-based):
Source order is **`needle$ INSTR haystack$`**, optionally **`FROM start%`** **where `start%` is a** 0-based **index in the haystack. Returns the** 0-based **index of the first match, or** `-1*` if not found.

```
LET pos% = "world" INSTR "Hello world"
LET pos% = "o" INSTR "Hello world" FROM 5
```

`REPLACE` - Replace substring:

```
LET new$ = "Hello world" REPLACE "world" WITH "EduBASIC"
```

Case and Whitespace

Several operators accept alternate spellings (`UPPER`, `LOWER`, sigil-suffixed trim forms, and others); see Appendix: Operator Reference.

`UCASE` - Convert to uppercase:

```
LET upper$ = UCASE "hello"
```

`LCASE` - Convert to lowercase:

```
LET lower$ = LCASE "WORLD"
```

`LTRIM` - Remove leading spaces:

```
LET trimmed$ = LTRIM "  text"
```

`RTRIM` - Remove trailing spaces:

```
LET trimmed$ = RTRIM "text  "
```

`TRIM` - Remove leading and trailing spaces:

```
LET trimmed$ = TRIM "  text  "
```

STR, VAL, HEX, and BIN

`STR` and `VAL` bridge between **values in memory** and **literal text** (source, Console EBON, DATA files). Formatting Numbers as Text introduced `STR` and `VAL` on numeric scalars; this section documents all scalar rules. Arrays and structures are covered in ch-arrays and ch-structures. Alternate spellings (`CSTR`, `STR$`, `CINT`, `HEX$`, `BIN$`, and others) are listed in Appendix: Operator Reference.

`STR`: Serialize any value to one EBON (EduBASIC Object Notation) line:

```
LET num% = 42
LET numStr$ = STR num%
LET piStr$ = STR 3.14159
LET complexStr$ = STR(3+4i)
LET quoted$ = STR "hello"
```

`STR` writes the same literal spelling you would type in the Code editor or Console. Structures, arrays, and all scalar tags can be written to DATA files and read back with `READ` / `VAL`.

`VAL`: Parse one EBON literal from a string (compact or multi-line document):

```
LET text$ = "123"
LET number% = VAL text$
```

```
LET decimal$ = "3.14"
LET value# = VAL decimal$
LET hexValue% = VAL "&HFF"
LET binValue% = VAL "&B1010"
LET complexValue& = VAL "3+4i"
```

The operand must be a **single literal expression** (no variables, no calls). Prefix `` `and `-` on numeric literals is allowed. Non-string operands are returned unchanged so `VAL+ can sit in mixed numeric expressions.

Display vs EBON: `PRINT` shows values in a human-readable form (strings print without surrounding quotes, so `PRINT "5"` and `PRINT 5` both display `5`). `CONSOLE` and bare expressions in the Console REPL use **EBON** literal spelling for each value, the same format as `STR`, so `CONSOLE "5"` displays `"5"` while `CONSOLE 5` displays `5`. `STR` produces EBON text for embedding in strings or files; `CONSOLE` writes the same spelling to the console scrollback.

`HEX` - Convert number to hexadecimal string:

```
LET hexStr$ = HEX 255
LET hexStr$ = HEX 10
LET hexStr$ = HEX -1
```

Converts an integer to its hexadecimal string representation (uppercase, no prefix). Negative numbers are represented using two's complement notation.

`BIN` - Convert number to binary string:

```
LET binStr$ = BIN 10
LET binStr$ = BIN 255
LET binStr$ = BIN -1
```

Converts an integer to its binary string representation. Negative numbers are represented using two's complement notation.

CHR and ASC

`CHR` - Code point to single-character string:

The operand is a numeric Unicode scalar value from `0` through `1114111` (`&H10FFFF`, all of Unicode). Isolated surrogate code unit values (`&HD800`–`&HDFFF`) are rejected. Values outside `0`…`&H10FFFF` are errors.

```
LET char$ = CHR 65
LET newline$ = CHR 10
LET slash$ = CHR &H2571
```

```
LET backslash$ = CHR 9586
LET grin$ = CHR &H1F600
```

ASC - First character to scalar code point:

Returns the Unicode scalar value of the string's first character (for characters outside the BMP, this is the full code point, not a UTF-16 surrogate half). Empty string yields `0`.

```
LET code% = ASC "A"
LET code% = ASC "a"
```

See Unicode and UTF-8 for how scalar values relate to UTF-8 storage and `LEN`.

Arrays

You met scalar types and sigils in Data Types, Values, and Variables. **Arrays** are composite values whose elements share one element type. Arrays are **0-based**: the first element is at index `0`. EduBASIC uses familiar `DIM`, `LET`, and bracket syntax; only the numeric index origin differs from some classic BASICs.

When you mean an **entire array** (not one element), use the `[]` rank suffix, for example `numbers%[]`. A bare `numbers%` can denote a scalar or an array name, so `[]` removes the ambiguity. Multi-dimensional whole-array references use a rank suffix that matches the dimension count: `matrix#[,]` for 2D, `cube%[,,]` for 3D, and so on.

Array Literals

Array literals are written using square brackets with comma-separated values:

```
[1, 2, 3, 4, 5]
["Alice", "Bob", "Charlie"]
[1.5, 2.7, 3.14]
[ ]
```

Rules:

- Array literals use square brackets `[]`
- Elements are separated by commas
- Elements can be numeric types (Integer, Real, Complex) or strings, but cannot be mixed
- Empty array literal `[ ]` creates an array with size 0
- Array literals create arrays with lower bound **0** (first element is index `0`)
- For numeric arrays, all elements are coerced to the most specific common type following the hierarchy: **Integer → Real → Complex**
- String arrays must contain only strings (no coercion)
- When assigning to a typed array variable, numeric elements are coerced to match the variable's type sigil

Examples:

```
LET numbers%[] = [1, 2, 3, 4, 5]
LET names$[] = ["Alice", "Bob", "Charlie"]
LET mixed#[] = [1.5, 2.7, 3.14]
LET empty%[] = [ ]
LET complex&[] = [1+2i, 3+4i, 5+6i]

' Type coercion in array literals
LET reals#[] = [0, 3.14]
LET cx&[] = [1, 3.14, 2+5i]

' Assignment coercion
LET x&[] = [1, 3.14]
LET y%[] = [5, 3.7]
```

Creating and Resizing Arrays

Array Creation

Arrays are created automatically when first assigned. No declaration is needed:

```
LET numbers%[] = [1, 2, 3, 4, 5]
LET names$[] = ["Alice", "Bob", "Charlie"]
```

Important: Undeclared arrays (arrays that haven't been assigned yet) are presumed to have size 0. You can assign to them or resize them with `DIM` before use.

Before the first assignment, reading an array name yields an empty array (`[ ]`, size 0), the same default noted in ch-data-types-values-and-variables#default-values.

Array Resizing

The `DIM` statement is used **only** for resizing arrays. Undeclared arrays are presumed to have size 0, so you can use `DIM` to resize them even if they haven't been assigned yet.

Syntax:

```
DIM arrayName[length]
DIM arrayName[length1, length2, ...]
DIM LOCAL arrayName[length]
DIM LOCAL arrayName[length1, length2, ...]
```

Each `length` is the **number of elements** on that axis. Valid indices run from `0` through `length - 1`. `DIM` does **not** accept `TO` ranges (`DIM a[1 TO 10]` is a parse error).

Rules:

- `DIM` sets or changes the size of an array; indices are always **0-based**
- If the array doesn't exist, `DIM` creates it
- If the array already exists, `DIM` resizes it: elements with indices `0 .. min(oldLength, newLength)-1` are **kept**; new slots are filled with the type's default value; excess elements are **discarded** when shrinking
- For multi-dimensional arrays, specify per-axis lengths separated by commas

Resize to a new size:

```
' 10 elements at indices 0..9
DIM numbers%[10]
LET numbers%[] = [1, 2, 3]
' Grow: first three elements preserved
DIM numbers%[20]
```

Multi-dimensional arrays:

```
DIM matrix#[5, 10]
```

Important (true multi-dimensional arrays):

- Multi-dimensional arrays in EduBASIC are **true N-dimensional arrays** (a single rectangular block), not jagged "arrays of arrays".
- Indexing a multi-dimensional array must use **comma-separated indices in a single bracket** (e.g. `matrix#[row%, col%]`).
- Index mapping uses **row-major order** (the **rightmost index varies fastest**).
- For multi-dimensional arrays, you must `DIM` the array first so it has well-defined bounds.
- element indexing/assignment (`a#[i, j]`, `LET a#[i, j] = ...`)
- total element count (`LEN matrix#[,]`)
- per-axis lengths (`EXTENTS matrix#[,]`)

Indexing and Modifying Arrays

These sections cover reading and writing elements, assigning through array paths, and the statements that grow, shrink, and reorder arrays in place.

Array Indexing and Bounds

Arrays are accessed using square brackets (indices **0-based**):

```
LET value% = numbers%[2]      ' third element
LET numbers%[4] = 100         ' fifth element
LET name$ = names$[1]         ' second string in the array
```

Bounds checking:

- Array indexing is **bounds-checked** at runtime.
- Reading or writing an element **outside the declared bounds** is a runtime error.

Arrays support multi-dimensional indexing:

```
LET value# = matrix#[1, 2]
LET matrix#[0, 4] = 42.5
```

Notes:

- `matrix#[2, 3]` is the supported form for 2D indexing (not `matrix#[2][3]`).
- Assignments to array elements use the `LET array[index] = value` and `LET array[i, j] = value` forms.

Slicing subranges (`[... TO ...]`), concatenating arrays with `, and unary *`LEN+ **on an entire array are** operators*; see Array Operators below.

Assigning to Arrays

You met `LET` and `LOCAL` for scalars in Data Types, Values, and Variables. The same keywords assign whole arrays, slices, and elements:

```
LET nums%[] = [10, 20, 30, 40, 50]
LET copy%[] = nums%[]
LET slice%[] = nums%[2 TO 4]    ' [30, 40, 50]
LET nums%[0] = 99
LOCAL temp%[] = copy%[]
```

Targets may combine structure fields and indices using the same path rules as in ch-structures.

Array Manipulation Statements

PUSH Statement:
Adds an element to the end of an array.

```
PUSH 10 INTO numbers%[]
PUSH "David" INTO names$[]
```

POP Statement:
Removes the last element from an array. Use `INTO` to assign the removed element to a variable, or omit `INTO` to discard it.

```
POP numbers%[]
POP numbers%[] INTO value%
POP names$[] INTO name$
POP numbers%[] INTO LOCAL temp%
```

SHIFT Statement:
Removes the first element from an array. Use `INTO` to assign the removed element to a variable, or omit `INTO` to discard it.

```
SHIFT numbers%[]
SHIFT numbers%[] INTO value%
SHIFT names$[] INTO name$
SHIFT numbers%[] INTO LOCAL temp%
```

UNSHIFT Statement:
Adds an element to the beginning of an array.

```
UNSHIFT 0 INTO numbers%[]
UNSHIFT "Alice" INTO names$[]
```

PLUCK Statement:
Removes one element from the middle of a 1D array, by value (first match) or by index (`PLUCK AT`). Use `INTO` to assign the removed element, or omit `INTO` to discard it. A value search that finds no match is a runtime error.

```
PLUCK "hawthorn" FROM inventory$[]
PLUCK 5 FROM numbers%[] INTO value%
PLUCK AT 2 FROM names$[]
PLUCK AT index% FROM items%[] INTO LOCAL item$
```

PLUCK ALL Statement:
Removes **every** element equal to `value` from a 1D array (same equality rules as `INDEXOF`). There is no `INTO` clause — multiple removed values cannot be assigned to one target. If no element matches, the array is unchanged (no error).

```
PLUCK ALL "hawthorn" FROM inventory$[]
PLUCK ALL 0 FROM numbers%[]
```

INSERT Statement:
Inserts a value **before** the element at a 0-based index. Valid indices are **0** through the current length (inclusive): index **0** prepends like `UNSHIFT`, index **length** appends like `PUSH`. An index greater than the length is a runtime error.

```
INSERT 9 AT 1 INTO numbers%[]
INSERT "Alice" AT 0 INTO names$[]
INSERT 42 AT LEN nums%[] INTO nums%[]
```

REVERSE Statement:
Reverses the element order of a 1D array **in place**.

```
REVERSE numbers%[]
REVERSE names$[]
```

Array Operators

These operators take an **array operand** on the left (`name[]` forms) where applicable.

INDEXOF, LASTINDEXOF, INCLUDES, and CONTAINS

Array search operations are implemented as binary operators:

INDEXOF Operator:
Finds the **0-based** index of the first occurrence of a value in an array. Returns **`-1`** if not found.

```
LET index% = numbers%[] INDEXOF 5
LET index% = names$[] INDEXOF "Bob"
```

LASTINDEXOF Operator:
Finds the **0-based** index of the **last** occurrence of a value in an array. Returns **`-1`** if not found.

```
LET index% = numbers%[] LASTINDEXOF 5
LET index% = names$[] LASTINDEXOF "Bob"
```

INCLUDES and CONTAINS Operators:
Checks if an array includes a value. Returns TRUE% (-1) if found, FALSE% (0) if not found. `INCLUDES` and `CONTAINS` are exact synonyms and can be used interchangeably.

```
IF numbers%[] INCLUDES 5 THEN
    PRINT "Found"
END IF
IF names$[] CONTAINS "Bob" THEN
```

```
    PRINT "Found"
END IF
```

JOIN Operator

Joins array elements into a string with a separator.

```
LET joined$ = names$[] JOIN ", "    ' "Alice, Bob, Charlie"
```

Array Slicing

Array slicing uses postfix `[start TO end]` on a 1D array operand. Bounds are **0-based** and **inclusive**: `nums%[2 TO 4]` copies the elements at indices 2, 3, and 4. Omit a bound with `...`: `[... TO n]` from the start, `[n TO ...]` through the end, and `[... TO ...]` for a full copy.

Unlike string `MID`, slice bounds must lie within the array; out-of-range indices are a runtime error. When the end index is less than the start index, the result is an empty array.

```
LET nums%[] = [10, 20, 30, 40, 50]
'              0   1   2   3   4

LET mid%[] = nums%[2 TO 4]       ' [30, 40, 50]
LET tail%[] = nums%[3 TO ...]     ' [40, 50]
LET head%[] = nums%[... TO 1]     ' [10, 20]
LET copy%[] = nums%[... TO ...]   ' [10, 20, 30, 40, 50]
```

Slicing creates a new array; the original is unchanged.
Slicing is only supported for **1D arrays**.

Array Concatenation

Arrays can be concatenated using the ``+ operator:

```
LET combined%[] = numbers%[] + more%[]
LET all%[] = [1, 2] + numbers%[] + [9, 10]
```

Concatenation creates a new array containing all elements from the left array followed by all elements from the right array.
Concatenation is only supported for **1D arrays**.

Array Length

Use unary `LEN` with an entire-array operand to get the total element count:

```
LET size% = LEN numbers%[]
```

For a 1D array, `LEN` is the number of elements (same as the declared length when the array is full). For multi-dimensional arrays, `LEN` returns the **total** flat element count (the product of all axis lengths). For numeric absolute value or complex magnitude on scalars, use `ABS`. String length uses the same **`LEN`** spelling; see String Length under Strings.

Array Extents

Use prefix unary `EXTENTS` with a whole-array operand to get per-axis lengths as a 1D integer array, in the same order as `DIM` and multi-index subscripts:

```
LET dims%[] = EXTENTS matrix#[,]    ' [5, 10] for DIM
matrix#[5, 10]
LET rows% = dims%[0]
LET cols% = dims%[1]

FOR i% = 0 TO dims%[0] - 1
    FOR j% = 0 TO dims%[1] - 1
        ' …
    NEXT j%
NEXT i%
```

For a 1D array, `EXTENTS numbers%[]` returns a single-element array `[n]` (equivalent to `[LEN numbers%[]]`). Use `dims%[0]` or `LEN` when you need a scalar length.

`EXTENTS` is only defined for array operands. It reads declared `DIMENSIONS` metadata for multi-dimensional arrays; undimensioned growable 1D arrays use the current element count.

Manipulation with Slicing and Concatenation

You can remove, insert, or replace spans by combining slices and `` `` `` `in a *LET` **`assignment to the whole array. The pattern is always`** `prefix middle suffix` **`: keep the elements before the edit, splice in the new span, then append the elements after the edit. For removing a`** `single* element by value or index, use the PLUCK+` statement (see Array Manipulation Statements).

Remove elements:

```
LET nums%[] = [10, 20, 30, 40, 50, 60]
'              0   1   2   3   4   5
```

```
' Drop indices 2..4 (30, 40, 50)
LET nums%[] = nums%[... TO 1] + nums%[5 TO ...]
' nums%[] is now [10, 20, 60]
```

Insert elements:

```
LET nums%[] = [10, 20, 30, 40, 50]
' Insert 99 and 88 before the element at index 2
LET nums%[] = nums%[... TO 1] + [99, 88] + nums%[2 TO ...]
' nums%[] is now [10, 20, 99, 88, 30, 40, 50]
```

Replace elements:

```
LET nums%[] = [10, 20, 30, 40, 50, 60]
' Replace indices 2..4 with 99 and 88
LET nums%[] = nums%[... TO 1] + [99, 88] + nums%[5 TO ...]
' nums%[] is now [10, 20, 99, 88, 60]
```

Comparing and Converting Arrays

These sections compare arrays for equality, serialize them to and from text, and exchange whole arrays or elements.

Comparing Arrays

Comparison operators work element-wise on arrays:

- Arrays are equal (=) if they have the same length and all corresponding elements are equal
- Arrays are not equal (<>) if they have different lengths or any corresponding elements differ
- Ordering operators (<, **>**, <=, >=) compare arrays lexicographically (element by element)
- If arrays have different lengths, the shorter array is considered less than the longer array
- If arrays have the same length, comparison proceeds element by element until a difference is found

Examples:

```
LET arr1%[] = [1, 2, 3]
LET arr2%[] = [1, 2, 3]
LET arr3%[] = [1, 2, 4]
```

```
LET arr4%[] = [1, 2]

IF arr1%[] = arr2%[] THEN
    PRINT "Equal"    ' TRUE%
END IF
IF arr1%[] <> arr3%[] THEN
    PRINT "Different"    ' TRUE%
END IF
IF arr1%[] < arr3%[] THEN
    PRINT "Less"    ' TRUE% (3 < 4)
END IF
IF arr4%[] < arr1%[] THEN
    PRINT "Shorter is less"    ' TRUE% (shorter array)
END IF
```

Numeric scalar comparisons are in Comparing Numbers under Numerical Operations.

STR and VAL with Arrays

`STR` turned integers and reals into text in Numerical Operations; here it serializes an entire array as one EBON literal:

```
LET numbers%[] = [1, 2, 3]
LET row$ = STR numbers%[]           ' "[1, 2, 3]"
LET copy%[] = VAL row$               ' Round-trip copy
```

`JOIN` builds a human-readable separated list; `STR` produces EBON spelling for files and `VAL`. See STR, VAL, HEX, and BIN under Strings for the full operator contract.

SWAP with Arrays

You met **`SWAP`** for scalar variables in SWAP Statement under Data Types, Values, and Variables. The same statement exchanges **array elements** and **whole arrays**, which is exactly what sorting, shuffling, and partition routines need without a temporary variable.

Syntax:

```
SWAP array1%[i], array2%[j]    ' Exchange two elements
SWAP first%[], second%[]       ' Exchange whole arrays
```

An element target uses the same indexing as **`LET`** (`[i]`, `[i, j]`, …); a whole-array target uses the `[]` form. As with scalars, you may prefix **`LOCAL`** on either operand independently.

Rules:

- Both operands must hold values of the **same type**. For arrays, the **element types** must match as well.
- Element targets may combine array indices with structure members (chained paths), following the same target rules as Assigning to Arrays.
- `SWAP` does not coerce types; mismatched types are a runtime error.

Examples:

```
DIM arr%[3]
LET arr%[0] = 30
LET arr%[1] = 10
LET arr%[2] = 20
SWAP arr%[0], arr%[2]    ' Exchange two elements

LET left%[] = [1, 2, 3]
LET right%[] = [9, 8, 7]
SWAP left%[], right%[]    ' Exchange whole arrays

' Order an adjacent pair, smallest first
IF arr%[0] > arr%[1] THEN
    SWAP arr%[0], arr%[1]
END IF
```

Structures

You met scalar types and sigils in Data Types, Values, and Variables. **Structures** are untyped records: a structure variable has **no type sigil**, and members are created when first assigned. Values are often built from Structure Literals in one expression; dot notation accesses and updates individual members.

Defining Structures

A structure can be written all at once as a literal or built up member by member.

Structure Literals

Structure literals are written using curly braces with comma-separated key-value pairs:

```
{ name$: "Alice", score%: 100, level%: 5 }
{ x%: 100, y%: 200 }
{ }
```

Rules:

- Structure literals use curly braces `{ }`
- Members are specified as `name: value` pairs
- Pairs are separated by commas
- Member names are identifiers (not strings)
- Member names may include an optional type sigil (%, #, $, &)
- Member names may include array brackets as part of the name (e.g. `scores%[]`, `matrix#[,]`)
- Values can be any data type (Integer, Real, Complex, String, Array, Structure)
- Empty structure literal `{ }` creates a structure with no members
- Structure literals can contain nested structures and arrays

Examples:

```
LET player = { name$: "Alice", score%: 100, level%: 5 }
LET point = { x%: 100, y%: 200 }
LET person = { name$: "John Doe", age%: 30 }
LET cfg = { width%: 640, height%: 480, title$: "Game" }
```

```
LET empty = { }

' Structure with array members
LET hero = { name$: "Alice", scores%[]: [100, 95, 87] }

' Structure with nested structures
LET person = { name: { first$: "John", last$: "Doe" } }
LET person.addr.city$ = "Springfield"
LET person.addr.zip% = 12345

' Arrays and nested structures
LET game.player.name$ = "Hero"
LET game.player.hp% = 100
LET game.foes%[] = [10, 15, 20]
```

Before the first assignment, reading a structure variable yields an empty structure (`{ }`, no members), consistent with the composite defaults in ch-arrays and Data Types, Values, and Variables.

Creating Structures

Structures can be created in two ways:

1. Using structure literals:

```
LET player = { name$: "Alice", score%: 100, level%: 5 }
```

2. By assigning values to members individually using dot member access:

```
LET player.name$ = "Alice"
LET player.score% = 100
LET player.level% = 5
```

Structure literals assign several members in one expression. Dot assignment assigns one member at a time and can extend an existing structure.

Working with Members

These sections read and update members with dot notation, including nested structures and array members.

Member Access

Structure members are accessed using the dot operator:

```
PRINT player.name$    ' Prints: Alice
LET player.score% += 10
IF player.level% > 10 THEN
    PRINT "Advanced player"
END IF
```

Structure Examples

```
' Create a structure for a point
LET point = { x%: 100, y%: 200 }

' Create a structure for a person
LET person = { name$: "John Doe", age%: 30 }

' Access structure members
PRINT person.name$, " age ", person.age%
LET person.age% += 1

' Structures can contain any data type
LET cfg = { width%: 640, height%: 480, title$: "Game" }
```

Nested Structures and Arrays

Structure members can themselves be arrays or structures, allowing for nested data structures:

```
' Structure with array members
LET hero = { name$: "Alice", scores%[]: [100, 95, 87] }

' Structure with nested structures
LET person = { name: { first$: "John", last$: "Doe" } }
LET person.addr.city$ = "Springfield"

' Access nested structure members
PRINT person.name.first$, " ", person.name.last$
PRINT person.addr.city$

' Access array members within structures
PRINT hero.scores%[0]
LET hero.scores%[1] = 98
```

Modifying and Comparing Structures

These sections assign whole structures and individual members, and compare structures for equality.

Assigning to Structures

You met `LET` and `LOCAL` for scalars in Data Types, Values, and Variables. The same keywords assign whole structures and individual members:

```
LET player = { name$: "Bob", score%: 100 }
LET player.score% += 10
LOCAL copy = player
LET copy.name$ = "Alice"
```

Targets may combine structure fields and array indices (for example `LET entity.pos%[i%].x% = n%` or `LOCAL entity.pos%[i%].x% = n%`) using the same path rules as in Assigning to Arrays under ch-arrays.

Comparing Structures

Structures support the equality operator (=). Two structures are equal if all of their members are equal (recursively, including nested structures and array members).

```
LET point1 = { x%: 100, y%: 200 }
LET point2 = { x%: 100, y%: 200 }

IF point1 = point2 THEN
    PRINT "Points are equal"    ' TRUE%
END IF

LET point3 = { x%: 100, y%: 201 }

IF point1 = point3 THEN
    PRINT "Equal"    ' FALSE% (y values differ)
END IF
```

Important Notes:

- The equality operator (=) is defined for structures
- Two structures are equal if all their members are equal (including nested structures and arrays)
- Inequality operators (<>, <, >, <=, >=) are **not** defined for structures
- Structure variables do not use type sigils (no %, #, $, or &)
- Structures are untyped, members can hold any data type
- Structure members can be arrays (using `[]` syntax)
- Structure members can be other structures (nested structures)
- Members are created automatically when first assigned
- No declaration is needed before using a structure

Converting and Swapping Structures

These sections serialize structures to and from EBON text and exchange whole structures or members.

STR and VAL with Structures

`STR` and `VAL` on scalars and arrays were developed in STR, VAL, HEX, and BIN and STR and VAL with Arrays. For structures, `STR` emits a pretty-printed EBON document (often multi-line); `VAL` parses it back:

```
LET player = { name$: "Alice", score%: 100, level%: 5 }
LET row$ = STR player
LET copy = VAL row$
```

Use `READ` and `WRITE` in `DATA` mode to load and save whole EBON documents from files; see File I/O. Scalar `VAL` rules (single literal, non-string passthrough) are defined in STR, VAL, HEX, and BIN — this section covers structure-shaped documents only.

Important Notes (structures recap):

- Structure variables do not use type sigils
- Members are created automatically when first assigned
- No declaration is needed before using a structure

SWAP with Structures

SWAP was introduced for scalars in SWAP Statement and extended to array elements and whole arrays in SWAP with Arrays. The same statement exchanges **structure members** and **whole structures**.

Syntax:

```
SWAP a.member, b.member
SWAP a, b
```

A member target uses dot paths (`.member`), optionally chained with further members and array indices, following the same target rules as Assigning to Structures. As with scalars and arrays, you may prefix **LOCAL** on either operand independently.

Rules:

- Member operands must hold values of the **same type**; `SWAP` does not coerce, and a mismatch is a runtime error.
- Whole-structure operands exchange entire records in one step (every member moves with it); both operands must be structures.
- Member targets may chain through nested structures and array indices, exactly as in Assigning to Structures.

Examples:

```
LET point = { x%: 10, y%: 20 }
SWAP point.x%, point.y%

LET hero = { name$: "Alice", hp%: 100 }
LET rival = { name$: "Bob", hp%: 80 }
SWAP hero, rival

LET party = { lead: { hp%: 30 }, backup: { hp%: 90 } }
SWAP party.lead.hp%, party.backup.hp%
```

Control Flow

Flow Control Concepts

This chapter pairs structured constructs with EduBASIC's truth rules; start here for the vocabulary the rest of the chapter assumes.

Structured vs. Unstructured Flow Control

EduBASIC supports structured constructs (`IF`, loops, `SUB`, and related forms) and label-based flow (`GOTO`).

Structured:

- `IF` / `THEN` / `ELSE` and `UNLESS`
- `SELECT CASE`
- `FOR` loops
- `WHILE` and `UNTIL` loops
- `WAIT` statement
- `DO` loops
- `SUB` procedures with parameters
- `TRY ... CATCH ... FINALLY` error handling

Unstructured:

- `GOTO`

Structured forms compose control flow without labels. `GOTO` transfers to `LABEL` names.

Note: EduBASIC does not have user-defined functions. Use `SUB` procedures with `BYREF` parameters to return computed values.

Conditions and Truth Values

Conditional statements evaluate **integer expressions**. Any non-zero value is **true**; `0` is **false**. The canonical true value is `TRUE%` (`-1`); the canonical false value is `FALSE%` (`0`). Bitwise operators (`AND`, `OR`, `NOT`, and related forms) are documented under Boolean Values and Bitwise Operators in Numerical Operations.

Comparison operators return `TRUE%` or `FALSE%`:

- Numeric scalars: Comparing Numbers under Numerical Operations
- Strings: String Comparison under Strings
- Arrays: Comparing Arrays under ch-arrays
- Structures: equality (=) only; see Comparing Structures under ch-structures

Use these operators in `IF`, `WHILE`, `UNTIL`, `SELECT CASE`, and loop conditions throughout this chapter.

Labels and Jumps

These sections cover unstructured flow: naming locations with labels and transferring control with `GOTO`.

Labels

Labels mark specific locations in code that can be targeted by `GOTO` statements. Unlike traditional BASIC which uses line numbers, EduBASIC uses descriptive labels.

Syntax:

```
LABEL labelName
```

Rules:

- Label names follow the same rules as variable names (alphanumeric, starting with a letter)
- Label names are case-insensitive
- Labels do **not** use type sigils
- Each label must be unique within its scope
- Labels can only appear at the beginning of a statement

Example:

```
LABEL StartProgram
PRINT "Beginning program..."

LABEL MainLoop
LET counter% = counter% + 1
IF counter% < 10 THEN
    GOTO MainLoop
```

```
END IF

LABEL EndProgram
PRINT "Program complete!"
```

GOTO Statement

The `GOTO` statement transfers control unconditionally to a labeled location.

Syntax:

```
GOTO labelName
```

Note: `GOTO` transfers control unconditionally. Loops and `IF` express repetition and branching with structured syntax instead of labels.

Example:

```
LET count% = 0

LABEL LoopStart
PRINT count%
LET count% = count% + 1
IF count% < 5 THEN
    GOTO LoopStart
END IF

PRINT "Done!"
```

Conditional Statements

These sections choose between branches: `IF` and its inverse `UNLESS` for two-way decisions, and `SELECT CASE` for multi-way dispatch.

IF, ELSEIF, ELSE, and END IF Statements

The `IF` statement executes code conditionally based on a boolean expression.

After the condition, the `THEN` keyword may be omitted when the clause header ends the line (typical for multi-line `IF` blocks). Canonical source (including reformatted lines in the editor) always includes `THEN`. The branch body is always on following lines through `END IF` (or the next `ELSE` / `ELSEIF` clause), not another statement on the same line as the header.

`UNLESS` uses the same optional-`THEN` parse rule, but canonical `UNLESS` headers **omit** `THEN` (see UNLESS, ELSE, and END UNLESS Statements).

Block `IF`:

```
IF condition THEN
    statements
END IF
```

The same block shape is accepted with `THEN` omitted on the header line, for example `IF condition` immediately followed by indented statements and `END IF`.

`IF-ELSE`:

```
IF condition THEN
    statements
ELSE
    statements
END IF
```

`IF-ELSEIF-ELSE`:

```
IF condition1 THEN
    statements
ELSEIF condition2 THEN
    statements
ELSEIF condition3 THEN
    statements
ELSE
    statements
END IF
```

Examples:

```
IF score% >= 90 THEN
    PRINT "Grade: A"
END IF

IF temperature# > 100 THEN
    PRINT "Water is boiling!"
    LET state$ = "gas"
END IF

IF age% < 13 THEN
    PRINT "Child"
```

```
ELSEIF age% < 20 THEN
    PRINT "Teenager"
ELSEIF age% < 65 THEN
    PRINT "Adult"
ELSE
    PRINT "Senior"
END IF
```

UNLESS, ELSE, and END UNLESS Statements

The `UNLESS` statement executes its body when the condition is false; it is equivalent to `IF NOT` condition.

After the condition, `THEN` may be omitted when the `UNLESS` header ends the line. Canonical source (including reformatted lines in the editor) **never** includes `THEN` on `UNLESS` headers.

Syntax:

```
UNLESS condition
    statements
END UNLESS

UNLESS condition
    statements
ELSE
    statements
END UNLESS
```

Examples:

```
UNLESS password$ = "secret"
    PRINT "Access denied!"
    EXIT
END UNLESS

UNLESS balance# >= price#
    PRINT "Insufficient funds"
ELSE
    LET balance# -= price#
    PRINT "Purchase complete"
END UNLESS
```

Note: `UNLESS condition` is exactly equivalent to `IF NOT condition`.

SELECT CASE, CASE, CASE ELSE, and END SELECT Statements

The `SELECT CASE` statement provides multi-way branching based on the value of an expression. This is EduBASIC's implementation of QuickBASIC's `SELECT CASE`.

Syntax:

```
SELECT CASE expression
    CASE value1
        statements
    CASE value2
        statements
    CASE value3, value4, value5
        statements
    CASE IS > value6
        statements
    CASE value7 TO value8
        statements
    CASE ELSE
        statements
END SELECT
```

`CASE` Clauses:

- **Single value:** `CASE 5` matches when expression equals `5`
- **Multiple values:** `CASE 1, 2, 3` matches when expression equals `1`, `2`, or `3`
- Available operators: =, <>, <, **>**, <=, >=
- **`CASE IS`** and numeric **`CASE … TO …`** require **integer, real, or string** values; **complex, array, and structure** selector values are rejected at runtime
- **Range:** `CASE 10 TO 20` matches when expression lies in the **inclusive** interval between the two bounds; the bounds may be written in **either order** (`CASE 20 TO 10` is equivalent)
- **Default:** `CASE ELSE` matches when no other case matches (optional)

Examples:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
    CASE 80 TO 89
        PRINT "B"
    CASE 70 TO 79
```

```
        PRINT "C"
    CASE 60 TO 69
        PRINT "D"
    CASE ELSE
        PRINT "F"
END SELECT
```

```
SELECT CASE command$
    CASE "QUIT", "EXIT", "Q"
        PRINT "Goodbye!"
        EXIT
    CASE "HELP", "?"
        CALL ShowHelp
    CASE "SAVE"
        CALL SaveGame
    CASE ELSE
        PRINT "Unknown command"
END SELECT
```

```
SELECT CASE age%
    CASE IS < 0
        PRINT "Invalid age"
    CASE 0 TO 2
        PRINT "Infant"
    CASE 3 TO 12
        PRINT "Child"
    CASE 13 TO 19
        PRINT "Teenager"
    CASE IS >= 20
        PRINT "Adult"
END SELECT
```

Loops

These sections cover EduBASIC's loop forms, from counted `FOR` loops to condition-driven `WHILE`, `UNTIL`, `WAIT`, and `DO` loops, plus `EXIT` and `CONTINUE` for altering loop flow.

FOR and NEXT Statements

The `FOR` loop iterates a counter variable through a range of values.

Syntax:

```
FOR var% = start% TO end% ⟨STEP step%⟩
    statements
NEXT ⟨var%⟩

FOR var# = start# TO end# ⟨STEP step#⟩
    statements
NEXT ⟨var#⟩

FOR LOCAL var% = start% TO end% ⟨STEP step%⟩
    statements
NEXT ⟨LOCAL var%⟩

FOR LOCAL var# = start# TO end# ⟨STEP step#⟩
    statements
NEXT ⟨LOCAL var#⟩
```

Rules:

- The loop variable must be numeric (integer or real)
- `STEP` is optional (defaults to `1`)
- `STEP` can be positive or negative
- The loop variable can be used inside the loop
- The `NEXT` statement can optionally specify the variable name for clarity

Examples:

```
FOR i% = 1 TO 10
    PRINT i%
NEXT i%
```

```
FOR count% = 0 TO 100 STEP 10
    PRINT count%
NEXT count%
```

```
FOR x# = 1.0 TO 0.0 STEP -0.1
    PRINT x#
NEXT x#
```

```
FOR row% = 1 TO 5
    FOR col% = 1 TO 5
        PRINT "*";
    NEXT col%
```

```
    PRINT
NEXT row%
```

WHILE and WEND Statements

The `WHILE` loop repeats while a condition is true, testing the condition before each iteration.

Syntax:

```
WHILE condition
    statements
WEND
```

Examples:

```
LET count% = 0
WHILE count% < 10
    PRINT count%
    LET count% += 1
WEND
```

```
LET input$ = ""
WHILE input$ <> "quit"
    PRINT "Enter command: ";
    INPUT input$
    PRINT "You entered: ", input$
WEND
```

```
WHILE NOT EOF fileHandle%
    READ line$ FROM fileHandle%
    PRINT line$
WEND
```

UNTIL and UEND Statements

The `UNTIL` loop repeats until the condition is true; it is equivalent to `WHILE NOT` condition.

Syntax:

```
UNTIL condition
    statements
UEND
```

Examples:

```
LET count% = 0
UNTIL count% >= 10
    PRINT count%
    LET count% += 1
UEND
```

```
LET input$ = ""
UNTIL input$ = "quit"
    PRINT "Enter command (or 'quit'): ";
    INPUT input$
    PRINT "You entered: ", input$
UEND
```

Note: `UNTIL condition` is exactly equivalent to `WHILE NOT condition`.

WAIT WHILE and WAIT UNTIL Statements

The **`WAIT`** statement has two forms: **`WAIT WHILE`** and **+WAIT UNTIL**. `Each form is a` **`single-line busy wait`**`: it re-evaluates an integer condition on every interpreter step and stays on that source line until the wait ends. Semantically, each form matches an empty` WHILE … WEND `or` UNTIL … UEND+ loop with no statements between the header and the closing keyword.

Syntax:

```
WAIT WHILE condition
WAIT UNTIL condition
```

Semantics:

- `WAIT WHILE condition`: waits while `condition` is **true** (non-zero), like `WHILE condition` with an immediate `WEND`.
- `WAIT UNTIL condition`: waits until `condition` is **true** (non-zero), like `UNTIL condition` with an immediate `UEND`.

The condition must evaluate to an **integer** (same rules as `WHILE` and `UNTIL`). There is no block body and no `WEND` / `UEND`.

Note: Execution spins on that line (the program counter does not advance) until another statement or the host changes variables so the condition allows progress. Use for short polls (for example waiting on `NOTES` or polling `INKEY$[]`); avoid tight infinite waits in lessons unless you intend to demonstrate responsiveness or breaking execution.

Examples:

```
' Wait until voice 1 has no notes (see NOTES)
WAIT WHILE NOTES 1 > 0
```

```
LET ready% = 0
' Runtime sets ready% when done
WAIT UNTIL ready%
PRINT "Ready."
```

DO and LOOP Statements

The DO loop provides flexible looping with conditions that can be tested at the beginning or end of the loop.

`DO WHILE` (condition tested at top):

```
DO WHILE condition
    statements
LOOP
```

`DO UNTIL` (condition tested at top):

```
DO UNTIL condition
    statements
LOOP
```

`DO-LOOP WHILE` (condition tested at bottom):

```
DO
    statements
LOOP WHILE condition
```

`DO-LOOP UNTIL` (condition tested at bottom):

```
DO
    statements
LOOP UNTIL condition
```

Unconditional `DO`:

```
DO
    statements
LOOP
```

Key Difference:

- Top-tested loops (`DO WHILE` / `DO UNTIL`) may never execute if the condition is initially `false` / `true`
- Bottom-tested loops (`DO … LOOP WHILE` / `DO … LOOP UNTIL`) always execute at least once

Examples:

```
LET password$ = ""
DO WHILE password$ <> "secret"
    PRINT "Enter password: ";
    INPUT password$
LOOP
```

```
DO
    PRINT "Enter a number (0 to quit): ";
    INPUT num%
    PRINT "You entered: ", num%
LOOP UNTIL num% = 0
```

```
PRINT "Press ESC to exit..."
DO
    IF INKEY$[] INCLUDES "ESC" THEN
        EXIT DO
    END IF
LOOP
```

EXIT Statement

Bare `EXIT` terminates the program immediately. Qualified forms exit a loop or `SUB` procedure.

Syntax:

```
EXIT
EXIT FOR
EXIT WHILE
EXIT DO
EXIT UNTIL
EXIT SUB
```

Examples:

```
IF criticalError% THEN
    PRINT "Fatal error occurred"
    EXIT
END IF
```

```
FOR i% = 1 TO 100
    IF numbers%[i%] = target% THEN
        PRINT "Found at position: ", i%
        EXIT FOR
    END IF
NEXT i%
```

```
DO
    PRINT "Enter value (negative to quit): ";
    INPUT value#
    IF value# < 0 THEN
        EXIT DO
    END IF
    LET sum# += value#
LOOP
```

Note: `END` closes structured blocks (`END IF`, `END SUB`, and related forms). Only bare `EXIT` halts the entire program.

CONTINUE Statement

The `CONTINUE` statement skips the rest of the current loop iteration and continues with the next iteration.

Syntax:

```
CONTINUE FOR
CONTINUE WHILE
CONTINUE DO
CONTINUE UNTIL
```

Examples:

```
FOR i% = 1 TO 10
    IF i% MOD 2 = 0 THEN
        CONTINUE FOR    ' Skip even numbers
    END IF
    PRINT i%    ' Only prints odd numbers: 1, 3, 5, 7, 9
NEXT i%
```

```
LET count% = 0
WHILE count% < 10
    LET count% += 1
    IF count% = 5 THEN
        CONTINUE WHILE    ' Skip printing 5
    END IF
    PRINT count%
WEND
```

```
DO
    PRINT "Enter number (0 to quit): ";
    INPUT num%
    IF num% = 0 THEN
        EXIT DO
    END IF
    IF num% < 0 THEN
        CONTINUE DO    ' Skip negative numbers
    END IF
    PRINT "Positive number: ", num%
LOOP
```

Note: `CONTINUE` is different from `EXIT`. `CONTINUE` skips to the next iteration of the loop, while `EXIT` exits the loop entirely.

Subroutines

These sections define procedures with `SUB`, pass arguments by value or by reference, and scope variables to a procedure's stack frame.

SUB and END SUB Statements

`SUB` defines a subroutine (procedure) that performs an action. Subroutines can accept parameters and modify variables through reference parameters or module-level variables.

Syntax:

```
SUB procedureName param1, param2, BYREF param3, ...
    statements
END SUB
```

Note: Parameters are passed **by value** by default. Prefix individual parameters with `BYREF` to pass by reference. Parentheses are not permitted in `SUB` declarations.

Calling a SUB:

The `CALL` statement invokes a subroutine procedure with arguments.

Syntax:

```
CALL procedureName arg1, arg2, ...
procedureName arg1, arg2, ...
```

Rules:

- The `CALL` keyword is optional, you can call a subroutine by name directly
- Parentheses are not permitted in `CALL` statements or direct subroutine calls
- Arguments are passed by position (first argument to first parameter, etc.)
- Arguments must match the parameter types (type sigils must match)
- Arguments are passed by value by default, unless the parameter is marked with `BYREF`

Examples:

```
CALL DrawBox 10, 5, "*"
DrawBox 20, 3, "#"    ' Same as CALL DrawBox 20, 3, "#"

SUB DrawBox width%, height%, char$
    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB
```

Rules:

- Parameters must include type sigils
- Parameters are passed **by value** by default (the subroutine receives a copy)
- Use `BYREF` keyword to pass **by reference** (allows modification of the original variable)
- Use `EXIT SUB` to return early
- `SUB`s can call other `SUB`s recursively

Parameter Passing

EduBASIC supports two parameter passing modes:

By Value (Default):

- The subroutine receives a **copy** of the argument
- Changes to the parameter do **not** affect the original variable
- This is the default behavior for all parameters

By Reference (with `BYREF`):

- The subroutine receives a **reference** to the original variable
- Changes to the parameter **do** affect the original variable when assigned with `LOCAL`
- Use the `BYREF` keyword before the parameter to enable this
- Inside the `SUB`, use `LOCAL` (not `LET`) to write back through a `BYREF` parameter; `LET` assigns module-level storage

Syntax:

```
SUB procedureName valueParam%, BYREF refParam#
    ' valueParam% is passed by value (default)
    ' refParam# is passed by reference (BYREF keyword)
END SUB
```

Example demonstrating the difference:

```
SUB TestParameters byValue%, BYREF byReference%
    LET byValue% = 999
    LOCAL byReference% = 999
END SUB
```

```
LET x% = 100
LET y% = 100
CALL TestParameters x%, y%
PRINT "x =", x%    ' Unchanged (by value)
PRINT "y =", y%    ' Changed (by reference)
```

Local Variables

Variables created with the LET statement inside a SUB are **module-level** (global). To create variables that are local to the current stack frame, use the LOCAL statement.

Syntax:

```
LOCAL variable = expression
LOCAL array[] = expression
```

Rules:

- Creates variables scoped to the current stack frame
- Inside a SUB: variables are local to that procedure's stack frame
- Outside any SUB: variables are local to the bottom-level stack frame
- Local variables are only accessible within their stack frame
- Local variables are destroyed when their stack frame is popped (when the SUB exits)
- Local variables do not conflict with module-level variables of the same name
- Local variables must still include type sigils
- Can create local scalars or arrays
- Local arrays are destroyed when the stack frame is popped

Examples:

```
SUB DrawBox width%, height%, char$
    LOCAL row%
    LOCAL col%

    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
```

```
END SUB

SUB CalcStats vals#[], n%, BYREF avg#, BYREF max#
    LOCAL sum# = 0
    LOCAL i%
    LOCAL buf%[] = [0, 0, 0]

    LOCAL max# = vals#[1]

    FOR i% = 1 TO n%
        LET sum# += vals#[i%]
        IF vals#[i%] > max# THEN
            LOCAL max# = vals#[i%]
        END IF
    NEXT i%

    LOCAL avg# = sum# / n%
END SUB
```

Examples:

```
SUB DrawBox width%, height%, char$
    LOCAL row%
    LOCAL col%

    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB

CALL DrawBox 10, 5, "*"
DrawBox 20, 3, "#"
```

```
SUB CalcStats vals#[], n%, BYREF avg#, BYREF max#
    LOCAL sum# = 0
    LOCAL i%

    LOCAL max# = vals#[1]
```

```
    FOR i% = 1 TO n%
        LET sum# += vals#[i%]
        IF vals#[i%] > max# THEN
            LOCAL max# = vals#[i%]
        END IF
    NEXT i%

    LOCAL avg# = sum# / n%
END SUB

DIM scores#[10]
' ... fill array ...
LET avg# = 0
LET max# = 0
CALL CalcStats scores#, 10, avg#, max#
PRINT "Average:", avg#, "Maximum:", max#
```

```
SUB InitializeGame
    ' These are module-level variables (global)
    LET score% = 0
    LET level% = 1
    LET lives% = 3
    PRINT "Game initialized!"
END SUB

CALL InitializeGame
```

Program Control

These sections halt the program with `EXIT` (see EXIT Statement), pause it for a duration with `SLEEP`, or stop for inspection with `BREAK` (like pressing **Pause**).

SLEEP Statement

The **SLEEP** statement pauses program execution for a specified number of milliseconds.

Syntax:

```
SLEEP milliseconds%
```

Examples:

```
PRINT "Starting in 3 seconds..."
SLEEP 1000    ' Wait 1 second
```

```
PRINT "2..."
SLEEP 1000    ' Wait 1 second
PRINT "1..."
SLEEP 1000    ' Wait 1 second
PRINT "Go!"
```

```
' Simple animation loop
FOR i% = 1 TO 10
    PRINT "Frame ", i%
    SLEEP 100    ' Wait 100ms between frames
NEXT i%
```

```
' Wait for user to read message
PRINT "Press any key to continue..."
SLEEP 2000    ' Give user 2 seconds to read
```

Rules:

- `milliseconds%` must be a non-negative integer
- The program pauses for the specified duration
- Other operations (like keyboard input or music playback) may still be processed during the sleep period

BREAK Statement

The **BREAK** statement acts as a program breakpoint: execution stops immediately, as though you pressed **Pause** on the debugger toolbar. Variables and the program counter are preserved so you can inspect state, then press **Continue** or **Step** to resume.

Syntax:

```
BREAK
```

Example:

```
FOR i% = 1 TO 100
    LET total% = total% + i%
    IF i% = 50 THEN
        BREAK
    END IF
NEXT i%
```

Rules:

- `BREAK` has no operands
- After a `BREAK`, the next statement to run is the line following `BREAK`
- Use `BREAK` during development to inspect variables at a chosen point in the program

Error Handling

EduBASIC provides structured error handling through `TRY … CATCH … FINALLY` blocks. Errors in EduBASIC are represented as strings, making error messages easy to create, compare, and display.

TRY, CATCH, FINALLY, and END TRY Statements

The `TRY … CATCH … FINALLY` statement provides structured exception handling. Errors are represented as strings.

Syntax:

```
TRY
    statements
CATCH errorVariable$
    statements
FINALLY
    statements
END TRY
```

Rules:

- The `TRY` block contains code that might raise an error
- The `CATCH` block executes if an error occurs in the `TRY` block
- When `errorVariable$` is present on `CATCH`, the error message is assigned to it as a string
- The `FINALLY` block always executes, whether an error occurred or not
- `FINALLY` is optional; you can use `TRY … CATCH … END TRY` without `FINALLY`
- `CATCH` is also optional; you can use `TRY … FINALLY … END TRY` to ensure cleanup without handling errors
- Errors are strings, so you can compare them, display them, or process them as needed

Examples:

```
TRY
    OPEN "data.txt" FOR TEXT READ AS file%
    READFILE content$ FROM "data.txt"
    CLOSE file%
CATCH error$
    PRINT "Error occurred: ", error$
FINALLY
    IF EXISTS "data.txt" THEN
        CLOSE file%
    END IF
END TRY
```

```
TRY
    LET result# = 10 / divisor%
    PRINT "Result: ", result#
CATCH error$
    IF error$ = "Division by zero" THEN
        PRINT "Cannot divide by zero"
    ELSE
        PRINT "Error: ", error$
    END IF
END TRY
```

```
' Try without catch (only finally for cleanup)
TRY
```

```
    OPEN "temp.txt" FOR TEXT OVERWRITE AS file%
    WRITE "data" TO file%
FINALLY
    CLOSE file%
END TRY
```

```
' Try without finally (only catch for error handling)
TRY
    LET value% = VAL userInput$
    PRINT "Parsed value: ", value%
CATCH error$
    PRINT "Invalid input: ", error$
END TRY
```

Error Propagation:

- If an error occurs in a `TRY` block and is not caught, or if `THROW` is called in a `CATCH` block, the error propagates to the enclosing `TRY` block (if any)
- If no enclosing `TRY` block exists, the program terminates with the error message

THROW Statement

The `THROW` statement raises (throws) an error. Errors are represented as strings.

Syntax:

```
THROW errorMessage$
```

Examples:

```
IF divisor% = 0 THEN
    THROW "Division by zero"
END IF
```

```
IF NOT EXISTS filename$ THEN
    THROW "File not found: " + filename$
END IF
```

```
SUB ValidateInput (value%)
    IF value% < 0 THEN
        THROW "Value must be non-negative"
    END IF
    IF value% > 100 THEN
        THROW "Value must not exceed 100"
    END IF
```

```
END SUB

TRY
    ValidateInput userValue%
CATCH error$
    PRINT "Validation failed: ", error$
END TRY
```

Rules:

- `THROW` immediately transfers control to the nearest `CATCH` block
- If no `CATCH` block exists, the program terminates with the error message
- Error messages are strings, so you can construct them dynamically
- `THROW` can be called from anywhere, including inside `SUB` procedures

Nested Error Handling:

```
TRY
    TRY
        OPEN "config.txt" FOR TEXT READ AS configFile%
        READFILE config$ FROM "config.txt"
    CATCH error$
        IF error$ = "File not found" THEN
            ' Try default config
            READFILE config$ FROM "default.txt"
        ELSE
            THROW error$    ' Re-throw other errors
        END IF
    END TRY
CATCH error$
    PRINT "Failed to load configuration: ", error$
    EXIT
END TRY
```

Part II

Console, Text, and File I/O

Console and Text I/O

EduBASIC provides a text output system that is separate from the 640×480 graphics system. The text system is rendered as an overlay on top of the graphics display, allowing you to combine text output with graphics operations. Text is displayed in a character grid, and you can control the position, color, and formatting of text output.

Note: All colors in EduBASIC use 32-bit RGBA format, where each component (Red, Green, Blue, Alpha) is 8 bits, allowing for 256 levels per channel and full transparency control.

Text Output

These statements write text to the console, mirror it to the host log, and clear the screen.

PRINT Statement

The `PRINT` statement outputs text and values to the text display. It can print scalars, arrays, or a combination of both.

Syntax:

```
PRINT expression1, expression2, ...
PRINT expression1, expression2, ...;
PRINT array[]
PRINT array[];
PRINT
PRINT expression1, expression2, ...; OPAQUE
```

Mixed Type Arguments:

The `PRINT` statement accepts expressions of any type (Integer, Real, Complex, String, Array). Each expression is converted to its string representation and emitted in source order.

Output Formatting:

- **Comma (`,`):** After a value, prints a **single space** before the next value
- **Semicolon (`;`):** After a value, prints **no** extra space before the next value
- **Trailing comma or semicolon:** Suppresses the final newline so the **next** `PRINT` continues on the same line
- **No separator after the last value:** Ends the line (adds a newline)
- **Empty `PRINT`:** Outputs a blank line (newline only)
- **Array:** When printing an array, all elements are printed in brackets with spaces after commas (e.g., `[1, 2, 3, 4, 5]`)
- **`OPAQUE`:** After the print list (including a trailing comma or semicolon), replaces text cell pixels instead of alpha-compositing over existing graphics

Type Conversion:

All value types are automatically converted to strings when printed:

- **Integer:** Decimal representation (e.g., `42` → `"42"`)
- **Real:** Decimal representation (e.g., `3.14` → `"3.14"`)
- **Complex:** Format `"real`imaginaryi" `or` "real-imaginaryi" `(e.g., 34i` → `"34i"+)`
- **String:** Printed as-is
- **Array:** All elements printed in brackets with spaces after commas (e.g., `[1, 2, 3]` → `"[1, 2, 3]"`)

Controlling Newlines:

- To avoid a newline at the end of output, end the `PRINT` statement with a **comma** or **semicolon** after the last item
- To output a blank line, use an empty `PRINT` statement (no arguments)

Examples:

Basic Printing:

```
PRINT "Hello, world!"
PRINT "Name: ", name$, " Age: ", age%
PRINT "X: ", x%, " Y: ", y%
```

Mixed Types:

```
LET count% = 10
LET price# = 19.99
LET name$ = "Widget"
LET z& = 3+4i

PRINT name$, count%, price#, z&
```

Print Arrays:

```
PRINT numbers%[]
PRINT scores%[]
```

Suppress Newline:

```
PRINT "Enter your name: ";
INPUT name$
```

Print Blank Lines:

```
PRINT
PRINT "Line 1"
PRINT
PRINT "Line 3"
```

Multiple Items on Same Line:

```
PRINT "Count: ", count%, "  ";
PRINT "Total: ", total#
```

String Concatenation Alternative:

You can also use string concatenation with the ``+ operator to build strings before printing. Spacing and formatting come from the strings you join:

```
' Using PRINT with commas (single space between items)
PRINT "Name: ", name$, "Age: ", age%
' Prints: Name: Alice Age: 25

' String concatenation (explicit spacing)
PRINT "Name: " + name$ + " Age: " + STR age%

LET msg$ = STR result# + " (" + STR count% + ")"
PRINT msg$
```

Commas vs concatenation:

- **Commas in PRINT:** Inserts a single space between values and converts each item to text
- **String concatenation:** Builds one string before printing; spacing comes from the strings you join

Special characters in string literals (`\n`, `\t`, `\r`, `\"`, `\\`): see String Literals under Strings.

CONSOLE Statement

The **`CONSOLE`** statement evaluates one or more expressions and prints them to the **Console** tab scrollback (not the **Output** panel). Use it in program code for debugging without cluttering normal `PRINT` output. In the Console REPL, typing a bare expression is equivalent to **`CONSOLE`** with one operand.

Syntax: Same comma and semicolon list rules as **`PRINT`** (see PRINT Statement); at least one expression is required.

Output formatting (EBON): Each value is serialized as one **EBON** literal, the same spelling as `STR` and DATA `WRITE` lines, not the display form used by `PRINT`:

- **`CONSOLE 5`** → `5`
- **`CONSOLE "5"`** → `"5"` (quoted string literal)
- **`PRINT "5"`** and **`PRINT 5`** → both 5 on the Output panel

Arrays and structures appear as bracket or brace literals (e.g. `[1, 2, 3]`, `{ name$: "Ada" }`). To build one human-readable line in the console, concatenate with `` `` `` and STR+ as needed.

Examples:

```
CONSOLE 1 + 2
CONSOLE x%, y%
CONSOLE "Value: " + STR value#
CONSOLE "5"          ' Console scrollback: "5"
CONSOLE 5            ' Console scrollback: 5
```

CLS Statement

The `CLS` statement clears the graphics screen.

Syntax:

```
CLS
CLS WITH backgroundColorExpr
```

Rules:

- Clears the entire graphics display (640×480 pixels)
- Fills every pixel with the current **background color** from `COLOR … BACKGROUND …` (same as the paper color used when the text window scrolls). At program start and IDE reset, that default is opaque black (`&H000000FF`). `CLS WITH &H00000000` (or `COLOR BACKGROUND` then `CLS`) still clears to transparent; the output view uses a black mat behind the canvas where alpha is zero
- With `WITH backgroundColorExpr`, clears to the color given by the expression, using the same rules as **`COLOR`**: 32-bit RGBA integers in `&HRRGGBBAA` format, or standard **CSS color names** in string values (case-insensitive), such as `CLS WITH "navy"`
- Does not affect the text display overlay
- After `CLS`, the next `TURTLE` begins with the turtle at the center, pen down, facing up
- Automatically switches to the output tab

Examples:

```
CLS

CLS WITH &H000033FF
CLS WITH "navy"

LET bg& = &HFF6600FF
CLS WITH bg&
```

INPUT Statement

The `INPUT` statement reads a value from the user and assigns it to a variable. Unlike traditional BASIC dialects, `INPUT` is separate from the prompt, allowing you to use `PRINT` for more flexible prompt formatting. `INPUT` can read into scalar variables or arrays.

Syntax:

```
INPUT variable%
INPUT variable#
```

```
INPUT variable$
INPUT variable&
INPUT array[]
INPUT LOCAL variable%
INPUT LOCAL variable#
INPUT LOCAL variable$
INPUT LOCAL variable&
INPUT LOCAL array[]
```

Rules:

- `INPUT` can read any data type (integer, real, string, complex)
- The variable's type sigil determines what type of value is expected
- The prompt is displayed separately using `PRINT` before the `INPUT` statement
- After the user enters a value and presses Enter, the value is assigned to the variable
- If the input cannot be converted to the variable's type, an error occurs
- For arrays, `INPUT` reads multiple values separated by commas, filling the array from index **0** onwards
- If more values are entered than the array size, excess values are ignored
- If fewer values are entered, remaining array elements are unchanged

Examples:

```
' Reading an integer
PRINT "Enter your age: ";
INPUT age%
PRINT "You are ", age%, " years old"

' Reading a real number
PRINT "Enter temperature: ";
INPUT temperature#
PRINT "Temperature is ", temperature#

' Reading a string
PRINT "Enter your name: ";
INPUT name$
PRINT "Hello, ", name$, "!"

' Reading a complex number
```

```
PRINT "Enter complex number (e.g., 3+4i): ";
INPUT z&
PRINT "You entered: ", z&

' Reading multiple values
PRINT "Enter X coordinate: ";
INPUT x%
PRINT "Enter Y coordinate: ";
INPUT y%
PRINT "Position: (", x%, ", ", y%, ")"

' Reading into an array
DIM scores%[5]
PRINT "Enter 5 scores (comma-separated): ";
INPUT scores%[]
PRINT "Scores: ", scores%[]

' Using LOCATE for formatted input
LOCATE ROW 9 COLUMN 0
PRINT "Name: ";
INPUT playerName$
LOCATE ROW 10 COLUMN 0
PRINT "Score: ";
INPUT score%
```

Note: `INPUT` is separate from the prompt. `PRINT` with a trailing semicolon keeps the cursor on the same line as the prompt; `LOCATE` positions the cursor before `INPUT`.

Cursor Position

These cover moving the text cursor and reading its current row and column.

LOCATE Statement

The `LOCATE` statement positions the text cursor at a specific row and column in the text display.

Syntax:

```
LOCATE ROW row%
LOCATE COLUMN column%
LOCATE ROW row% COLUMN column%
```

Rules:

- Row and column are 0-based (row 0, column 0 is the top-left corner). At least one of `ROW` or `COLUMN` must be present.
- When only `ROW` is given, the current column is preserved; when only `COLUMN` is given, the current row is preserved.
- When both are given, `ROW` and `COLUMN` clauses may appear in either order in source text; canonical spelling (including editor reformatted lines) is always `LOCATE ROW … COLUMN ….`
- The text display has a fixed character grid width (**80 columns**). Row count depends on line spacing: **30 rows** with line spacing off (default), **24 rows** with line spacing on (see SET LINE SPACING Statement)
- Valid row range: **0–29** with line spacing off, or **0–23** with line spacing on; valid column range: **0–79**
- After `LOCATE`, subsequent `PRINT` statements output at the specified position

Examples:

```
LOCATE ROW 9 COLUMN 19
PRINT "This text appears at row 9, column 19"

LOCATE ROW 0 COLUMN 0
PRINT "Top-left corner"

FOR row% = 0 TO 9
    LOCATE ROW row% COLUMN row%
    PRINT "*"
NEXT row%
```

CRSRLIN% and CRSRCOL%

`CRSRLIN%` and `CRSRCOL%` are nullary **integer** operators that read the current text cursor **row** and **column** on the shared `PRINT` / `LOCATE` grid.

Syntax: `CRSRLIN% · CRSRCOL%`

Rules:

- Both return **0-based** indices, consistent with `LOCATE` and the rules in the LOCATE Statement section (row `0`, column `0` is the top-left of the text grid; valid row and column ranges are the same as for `LOCATE`).
- The values reflect the cursor position where the next character from `PRINT` will appear, after the last `LOCATE`, `PRINT` (including semicolon continuations and newlines), and related text movement the runtime has applied.
- After **`CLS`**, the output surface is cleared and the text cursor is homed to **row 0, column 0**; in that case `CRSRLIN%` and `CRSRCOL%` both read **0** (until further output moves the cursor). The same home position applies when other operations that clear the text surface and reset the cursor do so in the way described for those statements (for example, changing line spacing).

Example:

```
LOCATE ROW 5 COLUMN 10
LET row% = CRSRLIN%    ' 5
LET col% = CRSRCOL%   ' 10
```

Color and Display Settings

These set text color and adjust how the console renders text: line spacing, wrapping, and scrolling.

COLOR Statement

The `COLOR` statement sets the global foreground and/or background color for both text and graphics operations. These colors are used by default, but can be overridden in individual statements when needed.

Syntax:

```
COLOR foregroundColor%
COLOR foregroundColor% BACKGROUND backgroundColor%
COLOR BACKGROUND backgroundColor%
```

Color Format:

- **32-bit RGBA integers**: Format `&HRRGGBBAA` (hexadecimal); always use this format in examples
- **Color names**: Standard CSS color names as strings (case-insensitive)
- Each component (R, G, B, A) ranges from 0–255
- Alpha channel controls transparency (0 = fully transparent, 255 = fully opaque)
- Color names use full opacity (alpha = 255) by default
- String expressions are automatically recognized as color names when they match a valid CSS color name

Text Usage:

- `COLOR foregroundColor%` sets the global foreground color for subsequent `PRINT` statements
- `COLOR foregroundColor% BACKGROUND backgroundColor%` sets both global foreground and background colors
- `COLOR BACKGROUND backgroundColor%` sets only the background color (foreground unchanged)
- Background color is only used for text output

Graphics Usage:

- Sets the global foreground color for subsequent graphics operations
- Affects `PSET`, `LINE`, `RECTANGLE`, `OVAL`, `TRIANGLE`, and other drawing operations
- Graphics statements can optionally override the global color using `WITH color%`
- Does not affect `PUT` operations (which use the sprite's stored colors with alpha blending)
- Only the foreground color is used for graphics (background parameter is ignored)

Common Colors:

```
LET black% = &H000000FF      ' Black (opaque)
LET white% = &HFFFFFFFF      ' White (opaque)
LET red% = &HFF0000FF        ' Red (opaque)
LET green% = &H00FF00FF      ' Green (opaque)
LET blue% = &H0000FFFF       ' Blue (opaque)
```

```
LET yellow% = &HFFFF00FF     ' Yellow (opaque)
LET transparent% = &HFFFFFF00 ' White (fully transparent)
```

Examples:

```
COLOR &HFF0000FF
PRINT "This is red text"

COLOR &HFFFFFF00 BACKGROUND &H000000FF
PRINT "Invisible text on black"

COLOR BACKGROUND &H000000FF
PRINT "Text with black background"

COLOR "red"
COLOR "blue" BACKGROUND "white"

COLOR &HFF0000FF
PSET (100, 100)

COLOR "red"
PSET (100, 100)

COLOR &H00FF00FF
LINE FROM (0, 0) TO (100, 100) WITH &HFF0000FF

LINE FROM (0, 0) TO (100, 100) WITH "red"
LINE FROM (0, 0) TO (100, 100) WITH "cornflowerblue"
```

Note: The `COLOR` statement sets global colors that persist until changed. The default text color is white on a transparent background. Graphics statements use the global foreground color by default, but can override it with `WITH color%`.

SET LINE SPACING Statement

Syntax:

```
SET LINE SPACING ON
SET LINE SPACING OFF
```

Text grid dimensions:

Program text uses **GNU Unifont** bitmap fonts. **Narrow** glyphs occupy one column (**8×16** graphics pixels); **wide** glyphs occupy two columns (**16×16** pixels). The text display grid dimensions depend on the line spacing setting:

- Width: 640 pixels ÷ 8 pixels per column = 80 columns
- Height: 480 pixels ÷ 16 pixels per text row = 30 rows
- Glyphs are drawn with no extra spacing between lines
- Width: 640 pixels ÷ 8 pixels per column = 80 columns (unchanged)
- Height: Each text row uses **20** pixels vertically: a **16**-pixel-high glyph (unchanged size; no scaling) plus **4** pixels of blank space before the next row. So `24 × 20 = 480` pixels, using the full display height.

Note: Switching line spacing **ON** or **OFF** clears the entire output surface and resets the text cursor to the top-left corner, because the row count and layout change.

Examples:

```
SET LINE SPACING OFF
LOCATE ROW 29 COLUMN 0
PRINT "Bottom row"

SET LINE SPACING ON
LOCATE ROW 23 COLUMN 0
PRINT "Bottom row with spacing"
```

SET TEXT WRAP Statement

Syntax:

```
SET TEXT WRAP ON
SET TEXT WRAP OFF
```

The text wrap setting controls whether output wraps when the cursor would move past the right edge of the text display.

Rules:

- **With text wrap OFF (default):** Further characters are forced onto the last column (narrow) or clipped at the right edge; wide glyphs may be clipped if they do not fully fit.
- **With text wrap ON:** When the next glyph does not fit, the cursor moves to the next line before printing. A narrow glyph may still be printed in column 79; a wide glyph wraps if it would extend past column 79.

Note: Wrapping is by **glyph** (one or two columns per Unifont glyph), not by "words" in the word-processor sense. Short lines are unaffected.

Examples:

```
SET TEXT WRAP OFF
PRINT "Long line truncated at column 80..."

SET TEXT WRAP ON
PRINT "Long line wraps at column 80 when enabled..."
```

SET SCROLL Statement

Syntax:

```
SET SCROLL ON
SET SCROLL OFF
```

When the text cursor is on the bottom row and a newline is printed (or the cursor advances to the line below), behavior depends on the scroll setting:

Rules:

- **With scroll ON (default):** The output scrolls up by one text row so new text appears on the bottom row, like a typical scrolling terminal.
- **With scroll OFF:** The output does not scroll; the cursor wraps to column **0** of the top row so further printing overwrites from the beginning of the screen.

Examples:

```
SET SCROLL ON
SET SCROLL OFF
```

SET FULLSCREEN Statement

Syntax:

```
SET FULLSCREEN ON
SET FULLSCREEN OFF
```

When FULLSCREEN is **ON**, the application layout dedicates more screen space to the Output panel (text console and graphics overlay) and hides peripheral UI chrome. When **OFF**, the standard editor layout returns. The setting persists for the session until changed.

Example:

```
SET FULLSCREEN ON
CONSOLE "Immersive output layout"
```

See also SET in the statement reference for all SET forms (LINE SPACING, TEXT WRAP, AUDIO, SCROLL, and FULLSCREEN).

HELP Statement

Syntax:

```
HELP topic
```

The HELP statement prints syntax reminders for a statement, command, or operator name. It is especially useful in the **Console** tab when you need a quick reminder without opening the appendices.

Examples:

```
HELP PRINT
HELP FOR
HELP EXTENTS
HELP +
```

Topics are case-insensitive. Use the same spelling as in source code (for example HELP INDEXOF, not a paraphrase).

Keyboard Input

EduBASIC provides non-blocking keyboard input through the INKEY$[] nullary operator. It evaluates to a **one-dimensional string array** listing every key **currently held down** (no queue: the runtime tracks a set of depressed keys). Multiple keys appear as multiple elements, including **chords** (for example Shift plus a letter).

INKEY$[] Nullary Operator

Syntax: INKEY$[]

Semantics:

- Each evaluation builds a **fresh** 1D `String` array snapshot of the depressed-key set.
- **Empty array** if no keys are pressed.
- **Order** follows the runtime set (insertion order among keys still held). Treat order as **unspecified** for program logic unless you only need membership.
- **Non-blocking:** returns immediately.
- **Membership tests:** use the array search operator `INCLUDES` (same as for other 1D string arrays). For example: `IF INKEY$[] INCLUDES "ARROWLEFT" THEN` (truthy when the key is held). `INCLUDES` yields integer `-1` if found, `0` if not.

Stable snapshot for combos: Evaluate once per "frame" if you need several tests to agree:

```
LET keys$[] = INKEY$[]
IF keys$[] INCLUDES "SHIFT" AND keys$[] INCLUDES "A" THEN
    ' Both held at the time of LET
END IF
```

Element values: Each array element is either a **single-character** string (e.g. `"a"`, `"A"`, `"1"`, `" "`) or a **special key name** (see Appendix: INKEY$[] Special Key Strings). On macOS, the Command (⌘) key is reported as `"CMD"`.

Examples:

```
' Game loop: ESC exits, arrows move
DO
    IF INKEY$[] INCLUDES "ESC" THEN
        EXIT DO
    END IF
    IF INKEY$[] INCLUDES "ARROWUP" THEN
        LET y% -= 1
    END IF
    IF INKEY$[] INCLUDES "ARROWDOWN" THEN
        LET y% += 1
    END IF
    IF INKEY$[] INCLUDES "ARROWLEFT" THEN
        LET x% -= 1
    END IF
    IF INKEY$[] INCLUDES "ARROWRIGHT" THEN
        LET x% += 1
    END IF
```

```
    ' ... game logic ...
LOOP

' Function keys (still use INCLUDES)
IF INKEY$[] INCLUDES "F1" THEN
    CALL ShowHelp
END IF
IF INKEY$[] INCLUDES "F2" THEN
    CALL SaveGame
END IF
IF INKEY$[] INCLUDES "ENTER" THEN
    CALL SelectOption
END IF
```

Note: For blocking input that waits for the user to press Enter, use the `INPUT` statement instead of polling `INKEY$[]`.

File I/O

EduBASIC file handles use UTF-8 (see Unicode and UTF-8 under Strings). When you `OPEN` a file you must pick exactly one **record format**: `TEXT` or `DATA`: and that choice stays in effect until `CLOSE`. You cannot use `TEXT` encoding on a handle opened `FOR DATA`, or vice versa. All paths are relative to the virtual disk tree (see The virtual disk). `DATA` files use the same literal syntax as source and the Console (see Literals under Data Types, Values, and Variables).

- `TEXT`: Human-readable **lines**. `READ` fills only **string** variables (one line per target); `WRITE` accepts only **string** expressions. Use `SEEK`, `LOC`, and `EOF` for byte-position navigation.
- `DATA`: One **EBON (EduBASIC Object Notation) document** per file: a single value (usually a structure or array), often pretty-printed across multiple lines. `READ` loads the entire file into exactly one variable; each `WRITE` replaces the entire file with one value. `SEEK` and `LOC` are not supported on `DATA` handles (`EOF` is). Files in this format often use the **.ebon** extension; drag-and-drop imports **.ebon** verbatim into the virtual disk (Host file import).

Independently of the format, you also choose how the file is opened for access: `READ`, `APPEND`, or `OVERWRITE` (see table-access-modes below). `DATA` mode does not support `APPEND`.

Note: File handles are integer identifiers. You must `OPEN` before `READ`/`WRITE`/`SEEK`/`EOF`/`LOC` on that handle (`SEEK` and `LOC` apply to `TEXT` handles only), and `CLOSE` when finished.

The virtual disk

The Disk tab and data files

Real programs separate **code** (the rules) from **data** (the content the rules act on). In EduBASIC, **`program.bas`** stays on the **Code** tab; everything else lives on the **virtual disk** under the **Disk** tab—a folder tree rooted at the disk name. Paths in **`OPEN`** and **`EXISTS`** are relative to that tree, not your computer's file system.

`program.bas` always appears in the tree and stays tied to the editor buffer. Drag host files onto the disk tree to import them. Supported types include **`.bas`** and **`.ebon`**; see host-import-types.

Each `.ebon` file holds one **top-level EBON value**—usually an array of structures. A program typically declares a matching module-level array, then a loader **`SUB`** reads the file into that array at startup. Editing data means editing the `.ebon` file on the **Disk** tab, not hunting through **`PRINT`** statements in code.

Selecting a file opens it in the **Disk** editor. **Text** mode edits UTF-8; **Hex** mode shows raw bytes. `.ebon` files are text-shaped EBON—use **Text** mode to inspect records before you run the program. That inspection step is part of file-backed design: you can see exactly what **`READ`** … **`FROM`** will load.

Host file import

Import (host → EduBASIC disk): Drag files from your host onto the disk tree. Supported types:

Host types	Virtual file	Treatment
`.txt`, **`.bas`**, **`.ebon`**, **`.sprite`**, **`.song`**	Same base name	Stored **verbatim** (bytes copied into the virtual disk).
`.png`, **`.jpg`**, **`.jpeg`**, **`.gif`**, **`.bmp`**, **`.webp`**	**`basename.sprite`**	Raster image converted to a **sprite** file for **`GET`** / **`PUT`**.

Unsupported extensions are **skipped**; the shell may summarize skips and failures after a batch drop. Import **overwrites** an existing virtual path of the same name—including root **`program.bas`** if you drop a **`.bas`** file onto the tree root—so treat drag-and-drop as intentional replacement.

Export: Drag a virtual file row to the host to download it. Exporting the disk project is not the same as an in-program **`SAVE`**—export copies files out of the IDE; **`SAVE`** writes a file the running program creates on the virtual disk.

Opening and Closing Files

OPEN Statement

The **`OPEN`** statement opens a file and assigns a handle to a variable. The full form has **two independent choices**:

1. **Record format**: `TEXT` or `DATA` (determines how `READ`/`WRITE` encode file content).
2. **Access mode**: `READ`, `APPEND`, or `OVERWRITE` (`APPEND` is not valid with `DATA`; see table-access-modes).

Syntax:

```
OPEN filename$ FOR TEXT|DATA accessMode AS fileHandle%
OPEN filename$ FOR TEXT|DATA accessMode AS LOCAL fileHandle%
```

Choosing **TEXT** vs **DATA**

	FOR TEXT …	FOR DATA …
What the file holds	Plain text lines (one line = one string value per READ/WRITE target)	One EBON document (single literal value for the whole file)
READ	Only string variables ($); string arrays read one line per element	Exactly one target variable; loads and parses the entire file (scalar, whole 1D array, or structure)
WRITE	Only string expressions; each target writes one UTF-8 line plus newline	Exactly one expression; replaces the entire file with one pretty-printed EBON document plus trailing newline
SEEK / LOC	Supported (byte cursor)	Not supported (runtime error)
Typical use	Logs, CSV-like text, exports meant for humans or editors	Structured saves, config/scene files (**.ebon**), sprite arrays (**.sprite**)

Access mode (**TEXT** only for **APPEND**; **DATA** supports **READ** and **OVERWRITE**):

Mode	Behavior
READ	Open file for reading only
APPEND	Open for writing; new bytes are written after existing content (TEXT only; DATA rejects APPEND)
OVERWRITE	Open for writing; existing content is discarded (or file created empty)

Rules:

- The file handle variable receives an integer identifier
- Files must be opened before any read/write operations on that handle
- Attempting to open a non-existent file in `READ` mode causes an error
- Opening a non-existent file in `APPEND` or `OVERWRITE` mode creates the file
- The handle stores the chosen `TEXT` or `DATA` format; all later `READ`/`WRITE` on that handle use that format

Examples:

```
OPEN "data.txt" FOR TEXT READ AS inputFile%
OPEN "output.txt" FOR TEXT OVERWRITE AS outputFile%
OPEN "log.txt" FOR TEXT APPEND AS logFile%
OPEN "rows.ebon" FOR DATA OVERWRITE AS outFile%
```

CLOSE Statement

The `CLOSE` statement closes an open file.

Syntax:

```
CLOSE fileHandle%
```

Examples:

```
OPEN "data.txt" FOR TEXT READ AS file%
' ... read operations ...
CLOSE file%
```

Reading from Files

READ Statement

The `READ` statement reads from an open file. In `TEXT` mode, each variable in the list consumes one line. In `DATA` mode, exactly **one** variable is allowed and the entire file is loaded as one EBON document.

Syntax:

```
READ ⟨LOCAL⟩ variable(, ⟨LOCAL⟩ variable ...⟩ FROM fileHandle%
```

Each target may optionally use its own `LOCAL` prefix (for example `READ line$, LOCAL count% FROM file%`).

TEXT mode (`OPEN … FOR TEXT …`):

- Each target must be a string variable (`$`). Each `READ` reads one UTF-8 line (delimiter: newline; carriage return is not stored in the string).
- For a string array `arr$[]`, one line is read per element, in order.

DATA mode (`OPEN … FOR DATA …`):

- Exactly **one** variable target per `READ` (comma-separated multiple targets are not allowed).
- The entire file is parsed as one EBON literal (pretty-printed multi-line documents are supported).
- Scalar types follow the variable's sigil; a whole 1D array replaces the target array (element type follows the array name's sigil; length comes from the document). Structures use member names that match EduBASIC field names, including sigils.
- After a successful `READ`, the handle cursor is at end of file; a second `READ` errors unless the file was rewritten on a writable handle.

Rules:

- At end of file before a complete line, an error occurs (check with `EOF` where appropriate)
- Trailing commas in the variable list are not allowed

Examples:

```
OPEN "data.txt" FOR TEXT READ AS file%

WHILE NOT EOF file%
    READ line$ FROM file%
    PRINT line$
WEND

CLOSE file%
```

```
OPEN "game.ebon" FOR DATA READ AS file%

READ gameConfig FROM file%

CLOSE file%
```

Writing to Files

WRITE Statement

The `WRITE` statement writes to an open file. In `TEXT` mode, each expression writes one line. In `DATA` mode, exactly **one** expression is allowed and each `WRITE` **replaces the entire file** with one EBON document.

Syntax:

```
WRITE expression⟨, expression ...⟩ TO fileHandle%
```

TEXT mode:

- Each expression must evaluate to a **string**. The file receives the UTF-8 text plus one newline (`\n`) per item.

DATA mode:

- Exactly **one** expression per `WRITE`.
- The value is serialized as a pretty-printed EBON document plus a trailing newline, replacing all prior file content.
- A second `WRITE` on the same handle overwrites the previous document (only the last value remains until the next `WRITE`).

Rules:

- Trailing commas in the expression list are not allowed

Examples:

```
OPEN "output.txt" FOR TEXT OVERWRITE AS file%

WRITE "Name: ", playerName$ TO file%
WRITE "Score: ", STR score% TO file%

CLOSE file%
```

```
OPEN "save.ebon" FOR DATA OVERWRITE AS file%

WRITE gameConfig TO file%

CLOSE file%
```

File Navigation

SEEK Statement

The SEEK statement positions the file pointer at a specific byte position. It applies to TEXT handles only; SEEK on a DATA handle raises an error.

Syntax:

```
SEEK position% IN fileHandle%
```

Rules:

- Position is always in bytes (0-based)
- Position 0 is the beginning of the file
- For text files, positions refer to UTF-8 byte positions
- Seeking past end of file is allowed (file will extend on write)

Examples:

```
OPEN "log.txt" FOR TEXT READ AS file%

SEEK 0 IN file%
READ line$ FROM file%

CLOSE file%
```

EOF Operator

The EOF operator is a unary operator that checks if the file pointer is at the end of the file.

Syntax:

```
EOF fileHandle%
```

Returns: Integer (0 = false, -1 = true)

Examples:

```
OPEN "data.txt" FOR TEXT READ AS file%

WHILE NOT EOF file%
    READ line$ FROM file%
    PRINT line$
```

```
WEND

CLOSE file%
```

LOC Operator

The `LOC` operator returns the current byte position in the file (`TEXT` handles only; `LOC` on `DATA` raises an error).

Syntax:

```
LOC fileHandle%
```

Returns: Integer (current byte position, 0-based)

Examples:

```
OPEN "log.txt" FOR TEXT READ AS file%

LET startPos% = LOC file%
READ line$ FROM file%
LET endPos% = LOC file%

CLOSE file%
```

EXISTS Operator

The `EXISTS` operator checks whether a file or directory path is present on the virtual disk (The virtual disk). Paths are relative to that tree, not your computer's file system.

Syntax: `EXISTS path$`

Returns: Integer (`FALSE%` / `0` if absent, `TRUE%` / `-1` if present).

Example:

```
IF EXISTS "data.txt" THEN
    READFILE content$ FROM "data.txt"
ELSE
    PRINT "File not found"
END IF
```

File Management

EduBASIC provides convenient statements for common file operations on the virtual disk (The virtual disk).

READFILE Statement

The `READFILE` statement reads an entire file into a string variable.

Syntax:

```
READFILE variable$ FROM filename$
READFILE LOCAL variable$ FROM filename$
```

Examples:

```
READFILE config$ FROM "config.txt"
PRINT config$

READFILE jsonData$ FROM "data.json"
' Process jsonData$
```

WRITEFILE Statement

The `WRITEFILE` statement writes an entire string to a file.

Syntax:

```
WRITEFILE expression TO filename$
```

Examples:

```
LET report$ = "Sales Report" + CHR 10 + "Total: $1000"
WRITEFILE report$ TO "report.txt"

WRITEFILE output$ TO "results.txt"
```

LISTDIR Statement

The `LISTDIR` statement lists files in a directory.

Syntax:

```
LISTDIR array$[] FROM path$
LISTDIR LOCAL array$[] FROM path$
```

Rules:

- Returns an array of filenames (strings)
- Indices are **0-based** (same as any other array)
- Includes files and subdirectories
- The array is created automatically if it doesn't exist

Examples:

```
LISTDIR files$[] FROM "."
FOR i% = 0 TO LEN files$[] - 1
    PRINT files$[i%]
NEXT i%

LISTDIR docs$[] FROM "/Users/name/Documents"
```

MKDIR Statement

The `MKDIR` statement creates a directory.

Syntax:

```
MKDIR path$
```

Examples:

```
MKDIR "backups"
MKDIR "/Users/name/data"
```

RMDIR Statement

The `RMDIR` statement removes an empty directory.

Syntax:

```
RMDIR path$
```

Examples:

```
RMDIR "temp"
RMDIR "/Users/name/old_data"
```

COPY Statement

The `COPY` statement copies a file.

Syntax:

```
COPY "source" TO "destination"
```

Examples:

```
COPY "data.txt" TO "backup.txt"
COPY "/source/file.bin" TO "/dest/file.bin"
```

MOVE Statement

The `MOVE` statement moves or renames a file.

Syntax:

```
MOVE "source" TO "destination"
```

Examples:

```
MOVE "oldname.txt" TO "newname.txt"
MOVE "/temp/file.txt" TO "/final/file.txt"
```

DELETE Statement

The `DELETE` statement deletes a file.

Syntax:

```
DELETE "filename"
```

Examples:

```
DELETE "temp.txt"
DELETE "/Users/name/old_data.bin"
```

File I/O Examples

Example: Reading and Writing Text:

```
OPEN "students.txt" FOR TEXT READ AS inputFile%
OPEN "grades.txt" FOR TEXT OVERWRITE AS outputFile%

WHILE NOT EOF inputFile%
    READ line$ FROM inputFile%
    ' Parse line$ to extract name$ and score%
    ' (parsing logic here)

    IF score% >= 90 THEN
        WRITE name$, "A" TO outputFile%
```

```
    END IF
WEND

CLOSE inputFile%
CLOSE outputFile%
```

Example: DATA mode (one EBON document per file):

```
OPEN "scores.ebon" FOR DATA OVERWRITE AS file%

LET scores%[] = [95, 87, 92]
WRITE scores%[] TO file%
CLOSE file%

OPEN "scores.ebon" FOR DATA READ AS file%
READ scores%[] FROM file%
CLOSE file%
```

Example: Replacing a DATA file on each WRITE:

```
OPEN "draft.ebon" FOR DATA OVERWRITE AS file%

WRITE { version%: 1, name$: "alpha" } TO file%
WRITE { version%: 2, name$: "beta" } TO file%

CLOSE file%
```

Part III

Multimedia

Graphics

EduBASIC provides a comprehensive graphics system for drawing shapes, sprites, and images. Text output and graphics share one **640×480** pixel surface: `PRINT`, `LOCATE`, and related commands rasterize into the same buffer as `PSET`, `LINE`, and other graphics statements (see Console and Text I/O).

Coordinate System:

- Graphics coordinates use **(0, 0) at the bottom-left corner** (mathematical convention)
- X increases to the right (0 to 639)
- Y increases upward (0 to 479)
- This is different from the **text grid**, which uses **(0, 0) at the top-left** for +LOCATE+ rows and columns

Color Format:

- All graphics operations use **32-bit RGBA** color format
- Format: `&HRRGGBBAA` (hexadecimal); always use this format in examples
- Each component (R, G, B, A) ranges from 0–255
- Alpha channel controls transparency (0 = fully transparent, 255 = fully opaque)

Coordinate Syntax:

- Graphics commands use parentheses to denote planar points: `(x, y)`
- This is an exception to EduBASIC's general rule that parentheses are only for grouping expressions
- Coordinates are always written as `(x%, y%)` or `(x#, y#)` in graphics commands

Color Usage:

- Colors are 32-bit RGBA integers in `&HRRGGBBAA` format
- The +COLOR+ statement sets global foreground and background colors
- Graphics statements use the global foreground color by default
- Graphics statements can override the global color using `WITH color%` when needed

Interior paint (`WITH` on filled regions):

- A **32-bit RGBA integer** (`&HRRGGBBAA`), or
- A **string** holding a standard **CSS color name** (case-insensitive, same as elsewhere; see Appendix: CSS Color Names), or
- A **two-dimensional integer array** (`sprite%[,]`) produced by **`GET`** or loaded from a **`.sprite`** file: each element is packed as `0xRRGGBBAA`; row `0` is the bottom row

of the sprite (see *GET* below). Use **`EXTENTS sprite%[,]`** for `[height, width]`.

- When **`WITH`** supplies a sprite array, the interior is filled by **tiling** that bitmap across the logical screen. The tiling **phase is locked to screen coordinates** with origin **(0, 0)** at the bottom-left, so patterns **align** across separate shapes and **`PAINT`** regions.
- For **outline-only** shapes (no **`FILLED`**), **`WITH`** must be a **solid** color (integer or color-name string); **sprite arrays are not valid** for strokes.
- **`PAINT`** replaces flooded pixels with the chosen solid color or with each sampled tile pixel (no alpha composite over the old pixel). **`FILLED RECTANGLE`** / **`OVAL`** / **`CIRCLE`** / **`TRIANGLE`** composite interior pixels like ordinary **`WITH`** colors (sprite samples use alpha the same way as solid RGBA).
- **`OPAQUE`** on **`PSET`**, **`LINE`**, shape statements, **`PUT`**, and **`TURTLE`** replaces destination pixels instead of alpha-compositing. Transparent colors write as-is (including alpha = 0).

Note: The text grid (80×30 character cells) is laid out on that surface alongside pixel graphics; coordinate systems differ, but both draw on the same output surface.

Pixels and Lines

These statements draw the simplest primitives: individual pixels and straight line segments.

PSET Statement

The `PSET` statement sets a single pixel to a color.

Syntax:

```
PSET (x%, y%) ⟨WITH color%⟩ ⟨OPAQUE⟩
```

Rules:

- Coordinates are in graphics pixels (0-based, bottom-left origin)
- X ranges from 0 to 639
- Y ranges from 0 to 479
- Uses global foreground color (set with +COLOR+) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format

Examples:

```
COLOR &HFF0000FF    ' Set global color to red
PSET (100, 200)     ' Red pixel (uses global color)

PSET (200, 200) WITH &HFFFFFFFF

FOR i% = 0 TO 639
    PSET (i%, 240) WITH &HFFFFFFFF
NEXT i%
```

LINE Statement

The `LINE` statement draws a line between two points.

Syntax:

```
LINE FROM (x1%, y1%) TO (x2%, y2%)
LINE FROM (x1%, y1%) TO (x2%, y2%) WITH color%
```

Rules:

- Draws a line from point (x1%, y1%) to point (x2%, y2%)
- Uses global foreground color (set with +COLOR+) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- For rectangles, use the +RECTANGLE+ statement instead

Examples:

```
COLOR &H00FF00FF    ' Set global color to green
LINE FROM (10, 10) TO (100, 50)

LINE FROM (50, 50) TO (150, 150) WITH &HFFFFFFFF
```

Shapes

These statements draw outlined or filled shapes: rectangles, ovals, circles, arcs, and triangles.

RECTANGLE Statement

The `RECTANGLE` statement draws a rectangle outline or filled rectangle.

Syntax:

```
RECTANGLE FROM (x1%, y1%) TO (x2%, y2%)
RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) ⟨WITH c%⟩ ⟨FILLED⟩
```

Rules:

- Draws a rectangle from point (x1%, y1%) to point (x2%, y2%)
- Default is outline only
- `FILLED` draws a filled rectangle
- Uses global foreground color (set with +COLOR+) by default
- `WITH` overrides the global color for the outline or the interior (see *Interior paint* under Graphics): solid RGBA integer, CSS color **string**, or **`GET`**-format sprite array when **`FILLED`**; solid integer or string only when outline-only
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFFFFFFFF    ' Set global color to white
RECTANGLE FROM (50, 50) TO (150, 150)

RECTANGLE FROM (200,200) TO (300,300) WITH &HFF0000FF FILLED

RECTANGLE FROM (100, 100) TO (200, 200) FILLED

GET tile%[,] FROM (0, 0) TO (7, 7)
RECTANGLE FROM (100, 100) TO (180, 140) WITH tile%[,] FILLED
```

OVAL Statement

The `OVAL` statement draws an oval (ellipse) outline or filled oval.

Syntax:

```
OVAL AT (x%, y%) RADII (radiusX%, radiusY%)
OVAL AT (x%, y%) RADII (rx%, ry%) ⟨WITH c%⟩ ⟨FILLED⟩
```

Rules:

- Center point is at (x%, y%)
- `radiusX%` and `radiusY%` are the semi-axes (half-width and half-height)
- Default is outline only
- `FILLED` draws a filled oval
- Uses global foreground color (set with +COLOR+) by default
- `WITH` overrides the global color (see *Interior paint* under Graphics): solid color or sprite array when **`FILLED`**; solid only when outline-only
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- A circle is a special case where `radiusX%` equals `radiusY%`

Examples:

```
COLOR &HFFFF00FF    ' Set global color to yellow
OVAL AT (320, 240) RADII (100, 100)

OVAL AT (100, 100) RADII (60, 60) WITH &H00FF00FF FILLED

OVAL AT (200, 200) RADII (80, 40) WITH &HFF00FFFF FILLED
```

CIRCLE Statement

The `CIRCLE` statement is a convenience alias for `OVAL` when width equals height (a circle).

Syntax:

```
CIRCLE AT (x%, y%) RADIUS radius%
CIRCLE AT (x%, y%) RADIUS radius% ⟨WITH c%⟩ ⟨FILLED⟩
```

Rules:

- Center point is at (x%, y%)
- Radius is a numeric expression (integer or real)
- Equivalent to `OVAL AT (x%, y%) RADII (radius%, radius%)`
- Default is outline only
- `FILLED` draws a filled circle
- Uses global foreground color (set with +COLOR+) by default
- `WITH` overrides the global color (see *Interior paint* under Graphics): solid color or sprite array when **`FILLED`**; solid only when outline-only
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFFFF00FF    ' Set global color to yellow
CIRCLE AT (320, 240) RADIUS 50

CIRCLE AT (100, 100) RADIUS 30 WITH &H00FF00FF FILLED

CIRCLE AT (200, 200) RADIUS 40 FILLED
```

ARC Statement

The `ARC` statement draws an arc (a portion of a circle) between two angles.

Syntax:

```
ARC AT (x%, y%) RADIUS r% FROM a0# TO a1#
ARC AT (x%, y%) RADIUS r% FROM a0# TO a1# ⟨WITH c%⟩
```

Rules:

- Center point is at (x%, y%)
- Radius is a numeric expression (integer or real)
- Angles are specified in radians
- `startAngle#` is the starting angle, `endAngle#` is the ending angle
- The arc is drawn counterclockwise from `startAngle#` to `endAngle#`
- Uses global foreground color (set with +COLOR+) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- Angles use standard mathematical convention: 0 radians points right (positive x-axis), increasing counterclockwise

Examples:

```
COLOR &HFFFF00FF    ' Set global color to yellow
ARC AT (320, 240) RADIUS 50 FROM 0 TO 3.14159
```

```
' Draw a quarter circle arc
ARC AT (100, 100) RADIUS 30 FROM 0 TO 1.5708 WITH &H00FF00FF
```

```
' Draw arc using degrees converted to radians
LET a0# = 45 * 3.14159 / 180
LET a1# = 135 * 3.14159 / 180
ARC AT (200, 200) RADIUS 40 FROM a0# TO a1#
```

Note: To draw a full circle, use +CIRCLE+ instead. `ARC` is for partial circles only.

TRIANGLE Statement

The `TRIANGLE` statement draws a triangle outline or filled triangle.

Syntax:

```
TRIANGLE (x1%, y1%), (x2%, y2%), (x3%, y3%) ⟨WITH c%⟩
TRIANGLE (x1%, y1%), (x2%, y2%), (x3%, y3%) FILLED
TRIANGLE (x1%, y1%), (x2%, y2%), (x3%, y3%) ⟨WITH c%⟩ FILLED
```

Rules:

- Draws a triangle with vertices at (x1%, y1%), (x2%, y2%), and (x3%, y3%)
- Default is outline only
- `FILLED` draws a filled triangle
- Uses global foreground color (set with +COLOR+) by default
- `WITH` overrides the global color (see *Interior paint* under Graphics): solid color or sprite array when **`FILLED`**; solid only when outline-only
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFFFFFFFF    ' Set global color to white
TRIANGLE (100, 100), (200, 200), (150, 250)

TRIANGLE (50,50), (150,50), (100,150) WITH &HFF0000FF FILLED

TRIANGLE (200, 200), (300, 200), (250, 300) FILLED
```

Fills and Sprites

These statements flood-fill bounded regions and capture or stamp rectangular bitmaps as sprites.

PAINT Statement

The `PAINT` statement fills a bounded area with a solid color or a repeating sprite pattern.

Syntax:

```
PAINT (x%, y%) ⟨WITH paintExpr⟩
```

Rules:

- Fills the area starting at point (x%, y%)
- Fills until it reaches a boundary (pixels that differ from the starting pixel's color)
- `WITH paintExpr` supplies the fill: **32-bit RGBA** integer, **CSS color name** string, or **`GET`**-format **sprite integer array** (see *Interior paint* under Graphics). Tiling uses global screen phase at origin **(0, 0)**.
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
PAINT (100, 100) WITH &HFF0000FF

PAINT (200, 200) WITH &H00FF00FF

PAINT (150, 150) WITH "gold"

GET pat%[,] FROM (0, 0) TO (3, 3)
PAINT (160, 160) WITH pat%[,]
```

GET Statement

The `GET` statement captures a rectangular region of the screen into a **two-dimensional integer array** (sprite matrix). There is no separate sprite type; pixels are packed as integers in `sprite%[row, col]`.

Syntax:

```
GET spriteArray%[,] FROM (x1%, y1%) TO (x2%, y2%)
GET LOCAL spriteArray%[,] FROM (x1%, y1%) TO (x2%, y2%)
```

Rules:

- Captures the rectangular region from (x1%, y1%) to (x2%, y2%) (inclusive corners; order of corners does not matter)
- **Replaces** the destination with a 2D integer array whose shape is **`EXTENTS`** **`sprite%[,] = [height, width]`**
- Layout: row `0` is the **bottom** row of the captured rectangle; columns increase to the right; each cell is one 32-bit color packed as **0xRRGGBBAA** (same as `PSET` / +COLOR+)
- Pre-`DIM` has no effect; `GET` assigns the correctly sized matrix
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- The same matrix can be passed to **`WITH`** on **`PAINT`** or on **`RECTANGLE`** / **`OVAL`** / **`CIRCLE`** / **`TRIANGLE`** when **`FILLED`** to tile a pattern (see *Interior paint* under Graphics)

Examples:

```
' Capture a 32×32 sprite
GET sprite%[,] FROM (100, 100) TO (131, 131)

' Capture entire 640×480 screen
GET screen%[,] FROM (0, 0) TO (639, 479)
```

PUT Statement

The `PUT` statement draws a sprite (from a `GET` **2D integer matrix**) onto the screen.

Syntax:

```
PUT spriteArray%[,] AT (x%, y%) ⟨OPAQUE⟩
```

Rules:

- Draws the sprite at position (x%, y%)
- The sprite's bottom-left corner is positioned at (x%, y%)
- Source must be a **2D integer array** from `GET` or a compatible **`.sprite`** file: **`EXTENTS sprite%[,] = [height, width]`**, pixels at `sprite%[row, col]` as *0xRRGGBBAA+
- Pixels are alpha-blended automatically using the alpha channel from the sprite data
- Transparent pixels (alpha = 0) are not drawn
- Semi-transparent pixels are blended with the background
- **`OPAQUE`** writes every sprite pixel directly, including transparent pixels
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- The same matrix can be used as **`WITH`** on **`PAINT`** or **`FILLED`** shapes for repeating fills (*Interior paint*)

Examples:

```
' Draw sprite with automatic alpha blending
PUT sprite%[,] AT (200, 150)

' Animate sprite
PUT player%[,] AT (x%, y%)
```

Turtle Graphics

Logo-style turtle drawing on the shared canvas, driven by a simple command string.

TURTLE Statement

The `TURTLE` statement provides Logo-style turtle graphics using a simple command string. The turtle draws on the same 640×480 canvas as other graphics primitives.

Syntax:

```
TURTLE command$
```

Rules:

- After `CLS` or when execution state is reset (e.g. loading a new program), the turtle starts at center (320, 240), facing up (north)
- Otherwise, the turtle's position, angle, and pen state persist across successive `TURTLE` statements until `CLS` or such a reset
- Drawing uses the current foreground color set by +COLOR+
- Pen is down by default (drawing enabled)

Turtle Commands:

See Appendix: Turtle Commands for the full command reference. Summary:

Command	Description
`FD n`	Move forward n pixels
`BK n`	Move backward n pixels
`RT n`	Turn right n degrees
`LT n`	Turn left n degrees
`PU`	Pen up (stop drawing)
`PD`	Pen down (resume drawing)
`HOME`	Return to center, facing up

Commands are **case-insensitive**.

Examples:

```
' Draw a square
TURTLE "FD 100 RT 90 FD 100 RT 90 FD 100 RT 90 FD 100"

' Draw a triangle
TURTLE "FD 100 RT 120 FD 100 RT 120 FD 100"

' Draw without lines (move only)
TURTLE "PU FD 50 PD FD 100"

COLOR &HFF0000FF    ' Red
TURTLE "FD 100"
COLOR &H00FF00FF    ' Green
TURTLE "RT 90 FD 100"
```

Graphics Examples

Example: Drawing a Simple Scene:

```
CLS WITH &H000033FF    ' Dark blue background

' Draw ground
RECTANGLE FROM (0, 0) TO (639, 50) WITH &H00FF00FF FILLED

' Draw sun
CIRCLE AT (550, 400) RADIUS 40 WITH &HFFFF00FF FILLED

' Draw house
RECTANGLE FROM (200,50) TO (400,200) WITH &HFF0000FF FILLED
RECTANGLE FROM (250,50) TO (350,150) WITH &HFFFFFFFF FILLED
```

Example: Sprite Animation:

```
' Capture sprite (shape set by GET; EXTENTS player%[,] gives
[height, width])
GET player%[,] FROM (0, 0) TO (31, 31)

LET x% = 100
LET y% = 100

' Animation loop
DO
    CLS

    PUT player%[,] AT (x%, y%)

    ' Update position
    LET x% += 2
    IF x% > 600 THEN
        LET x% = 0
    END IF

    ' Small delay
    FOR i% = 1 TO 1000
    NEXT i%
LOOP
```

Audio

EduBASIC provides audio using Music Macro Language (MML) for note control and SpessaSynth for sound generation. Playback uses a **SoundFont** (sample-based General MIDI bank) shipped with the app (GeneralUser GS). Voices use standard General MIDI instrument program numbers (**1–128**). The first time audio runs in a session, the browser loads the sound bank; wait until loading finishes before expecting sound.

General MIDI / Audio Overview

Each voice's **instrument** (GM program **1–128**) selects its timbre from the loaded sound bank. You set the instrument with the `VOICE` statement and play notes with the `PLAY` statement using MML.

How Audio Works:

- **MML** controls which notes to play, timing, and velocity for all voices (see **MML Syntax Reference** below and Appendix: MML Syntax)
- **Instrument** (set by `VOICE voice INSTRUMENT instrument`) determines the timbre (piano, guitar, strings, etc.)
- The `VOICE` statement assigns a General MIDI instrument (by number or name) to a voice
- The `PLAY` statement plays MML sequences on a voice
- **Voice initialization:** Voices (**1–16**, matching MIDI channels **1–16**) start with a default instrument until configured with `VOICE`

General MIDI Instruments:

- **Program numbers 1–128** map to standard GM instruments (e.g. **1** = Acoustic Grand Piano, **41** = Violin, **57** = Trumpet)
- Instruments are grouped into families: Piano (**1–8**), Chromatic Percussion (**9–16**), Organ (**17–24**), Guitar (**25–32**), Bass (**33–40**), Strings (**41–48**), Ensemble (**49–56**), Brass (**57–64**), Reed (**65–72**), Pipe (**73–80**), Synth Lead (**81–88**), Synth Pad (**89–**

96), Synth Effects (**97–104**), Ethnic (**105–112**), Percussive (**113–120**), Sound Effects (**121–128**)

- You can specify the instrument by **number** (e.g. `VOICE 1 INSTRUMENT 1`) or by **name** (e.g. `VOICE 1 INSTRUMENT "Acoustic Grand Piano"`). Names are case-insensitive (see Appendix: General MIDI Instruments).

Note Numbers:

- Note numbers use the MIDI standard: 0–127
- Middle C (C4) is note number 60
- Lower numbers are lower pitches, higher numbers are higher pitches

Audio Setup

These configure the synthesizer before playback: selecting the audio engine and the percussion channel and drum kits.

SET AUDIO Statement

Syntax:

```
SET AUDIO ON
SET AUDIO OFF
```

The audio enable setting controls whether synthesizer output is produced.

Rules:

- **With audio ON (default):** All audio output is enabled. `PLAY`, `VOLUME`, and `TEMPO` statements work normally.
- **With audio OFF:** All audio output is disabled. `PLAY` statements are ignored, and no sound is produced.

Note: When audio is OFF, `PLAY` statements execute without error but produce no sound. The `NOTES` operator still returns valid values for voices that have been configured, but no actual audio playback occurs.

Examples:

```
SET AUDIO ON             ' Enable audio output (default)
PLAY 1, "CDEFGAB C"

SET AUDIO OFF            ' Disable audio output
PLAY 1, "CDEFGAB C"      ' No sound produced, but no error
```

Percussion and Drum Kits

General MIDI describes **two** common ways to get percussion:

- **Chromatic (pitched) percussion**: These are normal **instruments** (GM **programs 1–128**), including the **Chromatic Percussion** family (programs **9–16**: Celesta, Glockenspiel, Marimba, and so on) and others such as **Timpani** (**48**). Set the timbre with `VOICE … INSTRUMENT` (number or **name** from Appendix: General MIDI Instruments), then play **letter notes** or **`N`** values as **pitch**, same as a piano or strings.

```
TEMPO 100
VOICE 1 INSTRUMENT "Marimba"
PLAY 1, "O4 L8 CDEG L4 A O5 C O4 B A G L2 E"
```

- **Drum kit (key-based)**: On a **drum kit** preset, each **MIDI note number** does **not** mean "that pitch" in the usual sense; it selects a **specific drum or cymbal** (for example **36** = Bass Drum 1, **38** = Acoustic Snare). In MML, use the **`N`** prefix with those numbers (e.g. `PLAY 10, "N36 N38 N42"` on **voice 10**, the usual GM drum **MIDI channel**). The industry-standard assignments for the default kit are listed in Appendix: General MIDI Standard Percussion Map.

```
TEMPO 120
VOICE 10 INSTRUMENT "Standard 1 Kit"
VOICE 11 INSTRUMENT "Standard 1 Kit"
DO
    PLAY 10, "L8 N36 R N38 R N36 R N38"
    PLAY 11, "L8 N42 R N42 R N42 R N42"
    WHILE NOTES 10 OR NOTES 11
    WEND
LOOP
```

Instrument names and kits: Quoted names first match **drum-kit** presets in the loaded SoundFont (for example **`"Standard 1 Kit"`** in GeneralUser GS). That puts the voice in **drum** mode so **`N`** note numbers follow the GM percussion map. Other quoted kit names are in Appendix: Drum Kits. If no drum preset matches, names resolve to **melodic** presets: **GM bank 0** (programs **1–128**) first, then **any other non-drum** name in the SoundFont (including auxiliary banks in Appendix: Auxiliary Melodic Banks), see Appendix: General MIDI Instruments for standard GM program names (wording may differ in the file, e.g. **Grand Piano** vs Acoustic Grand Piano). **`INSTRUMENT`** with a **numeric** program always selects a **melodic** patch on **bank 0** (program **1** = Acoustic Grand Piano); pass a **string** kit name for drum mode.

Channels: EduBASIC **voices 1–16** correspond to **MIDI channels 1–16**. **Voice 10** is the usual GM percussion channel. Additional drum layers can use **11**, **12**, and so on.

Which preset is melodic or drum is determined by **INSTRUMENT**, not by the voice index alone.

Playback Control

These set tempo and volume, assign instruments to voices, and play notes and music.

TEMPO Statement

The TEMPO statement sets the playback tempo for MML music.

Syntax:

```
TEMPO beatsPerMinute%
```

Rules:

- Sets the tempo in beats per minute (BPM)
- Affects all subsequent PLAY statements using MML
- Default tempo is typically 120 BPM
- Tempo persists until changed by another TEMPO statement

Examples:

```
TEMPO 120    ' Set to 120 BPM (moderate tempo)
TEMPO 60     ' Set to 60 BPM (slow)
TEMPO 180    ' Set to 180 BPM (fast)
```

VOLUME Statement

The VOLUME statement sets the global volume for all audio output.

Syntax:

```
VOLUME volumeLevel%
```

Rules:

- Sets the global volume level (**0–100**)
- 0 = silent, 100 = maximum volume
- Default volume is **100**
- Affects all voices regardless of their configuration
- Volume persists until changed by another `VOLUME` statement
- Individual voice velocity (V in MML) is relative to the global volume

Examples:

```
VOLUME 100    ' Maximum volume (default)
VOLUME 50     ' Half volume
VOLUME 25     ' Quarter volume
VOLUME 0      ' Silent
```

VOICE Statement

The `VOICE` statement assigns a General MIDI instrument to a voice. The instrument determines the timbre (sound character) when the voice plays MML via `PLAY`.

Syntax:

```
VOICE voice% INSTRUMENT instrumentNumber%
VOICE voice% INSTRUMENT instrumentName$
```

Rules:

- `voice%` identifies the voice (**1–16**, matching MIDI channels **1–16**). Up to 16 voices are available.
- **Instrument by number:** `INSTRUMENT instrumentNumber%` uses a General MIDI **melodic** program number (**1–128**). For example, **1** = Acoustic Grand Piano, **41** = Violin, **57** = Trumpet. Numbers do **not** select drum kits.
- **Instrument by name:** `INSTRUMENT instrumentName$` uses a string matched to the loaded SoundFont. **Drum-kit** preset names are tried first (e.g. **`"Standard 1 Kit"`** with GeneralUser GS; see Appendix: Drum Kits). Otherwise the name resolves to a **melodic** preset: **bank 0** (programs **1–128**) first, then other non-drum

presets (e.g. auxiliary GeneralUser banks in Appendix: Auxiliary Melodic Banks), see Appendix: General MIDI Instruments. Names are case-insensitive.

- Voices retain their instrument until changed by another `VOICE` statement.
- Voices can be used in `PLAY` immediately; unconfigured voices use a default instrument.

Examples:

```
' Set voice 1 to Acoustic Grand Piano (GM program 1)
VOICE 1 INSTRUMENT 1

' Set voice 2 to Electric Guitar by program number (28)
VOICE 2 INSTRUMENT 28

' Set voice 3 to Violin by name
VOICE 3 INSTRUMENT "Violin"

' Set voice 4 to Synth Lead by name
VOICE 4 INSTRUMENT "Lead 1 (square)"

PLAY 1, "CDEFGAB C"    ' Piano
PLAY 2, "CEG"         ' Guitar chord
PLAY 3, "N60 N64 N67"  ' Violin
```

PLAY Statement

The `PLAY` statement plays music or sound on a specific voice. **All `PLAY` statements play music in the background**, execution continues immediately without waiting for the music to finish.

Syntax:

```
PLAY voice%, mml$
```

Rules:

- `voice%` identifies the voice (**1–16**) to play on. Use `VOICE` to set the instrument for that voice.
- `mml$` contains the Music Macro Language sequence (string expression)
- MML controls the frequency (notes), timing, and velocity for the voice
- The voice's timbre is determined by the General MIDI instrument set with `VOICE voice INSTRUMENT instrument`
- Each `PLAY` statement plays on the specified voice
- Multiple voices can play simultaneously
- **All `PLAY` statements execute asynchronously in the background**, the program continues immediately
- Velocity in MML (`V` command) is relative to the global `VOLUME` setting
- Each voice has a **playback queue** of up to **65,536 entries**. Each MML token (notes, rests, and commands such as `L`, `O`, `V`, `MN`, `ML`, `MS`) counts as one entry. If a `PLAY` would cause the queue for that voice to exceed 65,536 entries, a **Music Overflow** error is raised.

MML (Music Macro Language)

These cover the Music Macro Language strings that `PLAY` interprets: the full syntax reference and worked examples.

MML Syntax Reference

Music Macro Language (MML) controls frequency (notes), timing, and velocity for all voices. MML is used within `PLAY` statements to specify musical sequences. Appendix: MML Syntax lists the same tokens in a compact table.

Notes:

- **Note Names:** Note letters may be upper or lower case: `C`, `D`, `E`, `F`, `G`, `A`, `B` or `c`, `d`, `e`, `f`, `g`, `a`, `b`
- **Sharps:** + or `#` after note name (e.g., C+ or `C#` = C sharp)
- **Flats:** - after note name (e.g., `B-` = B flat)
- **Note Numbers:** `N` followed by MIDI note number (0–127), where 60 = Middle C (C4)

Octaves:

- `O` followed by number (0–9) sets the octave for subsequent notes
- **>** raises the octave by one (e.g. after C in octave 4, **>** then plays the next note in octave 5)
- < lowers the octave by one
- Default octave is typically 4 (middle octave)
- Higher numbers = higher octaves

Note Length:

- `L1` = whole note
- `L2` = half note
- `L4` = quarter note (common default)
- `L8` = eighth note
- `L16` = sixteenth note
- `L32` = thirty-second note
- Individual notes can override length: `C4` = C note with length 4
- **Dotted notes:** One or more `.` after optional length digits extend duration in the usual way: one dot adds half of the base note's value, a second dot adds half of what the first dot added, and so on (e.g. `L4 C.` = dotted quarter using the default length;

`C4.` = dotted quarter with explicit length; `C4..` = double-dotted quarter). The same rules apply to `R` rests and to `N` MIDI notes.

Rests:

- `R` followed by length creates a rest (silence)
- Example: `R4` = quarter note rest

Velocity:

- `V` followed by number (0–127) sets the velocity (loudness) for subsequent notes
- Default velocity is typically 64 (50%)
- Velocity is relative to the global `VOLUME` setting
- Velocity persists until changed by another `V` command, including across separate `PLAY` statements on the same voice (the same applies to current **octave** after `O` / **`>`** / `<`, **default length** after `L`, and **articulation** after `MN` / `ML` / `MS`)
- Higher velocity values produce louder notes (within the limits of the global volume)
- Example: `V100 C4 D4 V64 E4` plays C and D at velocity 100 (louder), then E at velocity 64 (softer)
- Example: `V127 C D E` plays all three notes at maximum velocity
- Example: `V0 C D E` plays all three notes silently (velocity 0)

Articulation:

- **`MN`**: Music normal (default). Each note sounds for **7/8** of its written duration; the remainder of the slot is silent before the next note or rest begins.
- **`ML`**: Music legato. Each note sounds for the **full** written duration (connected, smooth).
- **`MS`**: Music staccato. Each note sounds for **3/4** of its written duration (short, detached).
- Written duration (`L`, per-note digits, dots) still controls **when** the next event starts; articulation only changes **how long** the note sounds within that slot.
- Default articulation is **`MN`** (QuickBASIC-compatible). **`ML`** connects notes without re-attacking each pitch (for example `PLAY 1, "ML CDEFGAB"`).
- Articulation persists until changed by another `MN`, `ML`, or `MS` command, including across separate `PLAY` statements on the same voice (the same applies to **octave**, **default length**, and **velocity**).

- Example (compare the three styles at L8):

```
PLAY 1, "L8 MN C D E F G A B"    ' Normal (7/8 gate)
PLAY 1, "L8 ML C D E F G A B"    ' Legato (full gate)
PLAY 1, "L8 MS C D E F G A B"    ' Staccato (3/4 gate)
```

Checking Remaining Notes:

- Use the NOTES operator to check how many entries remain in a voice's playback queue
- Syntax: NOTES voice%
- Returns the number of entries (notes, rests, and unprocessed MML commands) remaining for that voice. Queues are per-voice and limited to 65,536 entries
- Returns 0 when the voice has finished playing all notes
- Useful for checking if a voice is still playing music
- Each O, L, V, MN, ML, MS, **>**, <, each note, and each rest counts as **one** queue entry

Examples:

```
' Set up voices with General MIDI instruments
VOICE 1 INSTRUMENT 1    ' Acoustic Grand Piano
VOICE 2 INSTRUMENT 28  ' Electric Guitar (clean)

TEMPO 120

' PLAY runs in background
PLAY 1, "CDEFGAB C"

' Play with velocity (V command)
PLAY 1, "V100 CDEF V64 GAB"

' Play with note numbers (MIDI)
PLAY 1, "N60 N62 N64 N65 N67"

' Check how many notes remain
LET notesLeft% = NOTES 1
IF notesLeft% > 0 THEN
    PRINT "Voice 1 still playing: ", notesLeft%, " notes"
END IF
```

```
' Play on different voice - same MML, different instrument
PLAY 2, "N60 L4"
```

MML Examples

Simple Scale:

```
VOICE 1 INSTRUMENT 1    ' Acoustic Grand Piano
TEMPO 120
PLAY 1, "CDEFGAB C"    ' MML controls notes
```

Song with Velocity:

```
VOICE 1 INSTRUMENT 1    ' Piano
TEMPO 100
' V command sets velocity (0-127), relative to global VOLUME
PLAY 1, "V80 C L4 D L4 E L2 V100 F L4 G L4 A L2"
```

Using Note Numbers:

```
VOICE 1 INSTRUMENT 1    ' Piano
TEMPO 120
' Play C major chord (C=60, E=64, G=67)
PLAY 1, "N60 N64 N67 L2"
```

Multiple Voices:

```
VOICE 1 INSTRUMENT 1    ' Piano for voice 1
VOICE 2 INSTRUMENT 28  ' Electric Guitar for voice 2
TEMPO 120

' Melody on voice 1 (plays in background)
PLAY 1, "CDEFGAB C"

' Harmony on voice 2
PLAY 2, "CEG CEG"

' Check remaining notes for both voices
LET notes0% = NOTES 1
LET notes1% = NOTES 2
PRINT "Voice 1: ", notes0%, " notes remaining"
PRINT "Voice 2: ", notes1%, " notes remaining"
```

Different instruments example:

```
VOICE 1 INSTRUMENT "Acoustic Grand Piano"
TEMPO 120

' Play melody at different pitches
PLAY 1, "N60 L4 N65 L4 N67 L4"
```

Audio Examples

Example: Simple Melody:

```
VOICE 1 INSTRUMENT 1    ' Acoustic Grand Piano
TEMPO 120
PLAY 1, "C D E F G A B C"    ' MML controls notes
```

Example: Song with Dynamics (Velocity):

```
VOICE 1 INSTRUMENT 1    ' Piano
TEMPO 100

' V command sets velocity (0-127), relative to global VOLUME
' Velocity affects loudness of individual notes
PLAY 1, "V64 C L4 V80 D L4 V96 E L4 V112 F L2"
```

Example: Different General MIDI Instruments:

```
' Create voices with different GM instruments
VOICE 1 INSTRUMENT 1    ' Acoustic Grand Piano
VOICE 2 INSTRUMENT 28   ' Electric Guitar (clean)
VOICE 3 INSTRUMENT 41   ' Violin

TEMPO 120

' Play melody on piano
PLAY 1, "N60 N62 N64 N65 N67 L4"

' Play chord on guitar
PLAY 2, "N40 L1"    ' Low chord

' Play sustained note on violin
PLAY 3, "N55 L1"    ' String texture
```

Example: Instruments by Name:

```
VOICE 1 INSTRUMENT "Acoustic Grand Piano"
VOICE 2 INSTRUMENT "Electric Guitar (clean)"

TEMPO 120
PLAY 1, "N60 L4 N65 L4 N67 L4"
PLAY 2, "CEG"
```

Example: Multiple Voices:

```
VOICE 1 INSTRUMENT 1    ' Piano
VOICE 2 INSTRUMENT 49   ' String Ensemble 1
TEMPO 100

' Melody and harmony
PLAY 1, "N60 N64 N67 L1"    ' C major chord on piano
PLAY 2, "N60 N64 N67 L1"    ' Same chord on strings
```

Part IV

Appendices

Appendix: Reserved Keywords

These words are **reserved** by the language (they cannot be used as **variable** or **structure member** names). Flat list **A–Z** (same set that EduBASIC treats as keywords).

ABS, ACOS, ACOSH, ADSR, ALL, AND, APPEND, ARC, AS, ASC, ASIN, ASINH, AT, ATAN, ATANH, AUDIO, BACKGROUND, BIN, BREAK, BYREF, CALL, CARG, CASE, CATCH, CBRT, CEIL, CHR, CLAMP, CIRCLE, CLOSE, CLS, COLOR, COLUMN, CONJ, CONSOLE, CONTAINS, CONTINUE, COPY, COS, COSH, CRSRCOL, CRSRLIN, DATE, DEG, DELETE, DIM, DO, E, ELSE, ELSEIF, END, ENDSWITH, EOF, EXISTS, EXIT, EXP, EXPAND, EXTENTS, FALSE, FILLED, FINALLY, FLOOR, FOR, FROM, GET, GOTO, HELP, HEX, IF, IMAG, IMP, IN, INCLUDES, INDEXOF, INKEY, INPUT, INSERT, INSTR, INSTRUMENT, INT, INTO, IS, JOIN, LABEL, LASTINDEXOF, LCASE, LEFT, LET, LEN, LINE, LISTDIR, LOC, LOCAL, LOCATE, LOG, LOG10, LOG2, LOOP, LTRIM, MID, MKDIR, MOD, MOVE, NAND, NEXT, NOR, NOT, NOTES, NOW, OFF, ON, OPEN, OPAQUE, OR, OVAL, OVERWRITE, PAINT, PI, PLAY, POP, PRESET, PRINT, PSET, PUSH, PUT, RAD, RADII, RADIUS, RANDOMIZE, READ, READFILE, REAL, RECTANGLE, REPLACE, REVERSE, RIGHT, RMDIR, RND, ROUND, ROW, RTRIM, SEEK, SELECT, SET, SGN, SHIFT, SIN, SINH, SLEEP, SPACING, SQRT, STARTSWITH, STEP, STR, SUB, SWAP, TAN, TANH, TEMPO, TEXT, THEN, THROW, TIME, TO, TRIANGLE, TRIM, TRUE, TRUNC, TRY, TURTLE, UCASE, UEND, UNLESS, UNSHIFT, UNTIL, VAL, VOICE, VOLUME, WAIT, WEND, WHILE, WITH, WRAP, WRITE, WRITEFILE, XNOR, XOR

Appendix: Statement Reference

This appendix is an alphabetical reference of **EduBASIC statements and commands** (including control-flow and graphics/audio/file forms) and **syntactic keywords** used in statement syntax. It does not list expression **operators** or mathematical forms; those are in Appendix: Operator Reference.

ARC

Type: Command (Graphics)
Syntax: `ARC AT (x%, y%) RADIUS radius% FROM startAngle# TO endAngle# ⟨WITH color%⟩`
Description: Draws an arc (a portion of a circle) between two angles. Angles are in radians, with 0 pointing right (positive x-axis), increasing counterclockwise. Uses global foreground color by default, or `WITH color%` to override. Coordinates use bottom-left origin (0,0).
Example:

```
ARC AT (320, 240) RADIUS 50 FROM 0 TO 3.14159
ARC AT (100, 100) RADIUS 30 FROM 0 TO 1.5708 WITH &H00FF00FF
```

BREAK

Type: Command (Control Flow)
Syntax: `BREAK`
Description: Acts as a program breakpoint: execution stops immediately, as though you pressed **Pause** on the debugger toolbar. Variables and the program counter are preserved so you can inspect state, then press **Continue** or **Step** to resume. After a `BREAK`, the next statement to run is the line following `BREAK`. Tutorial: BREAK Statement under Control Flow.
Example:

```
FOR i% = 1 TO 100
    LET total% = total% + i%
    IF i% = 50 THEN
        BREAK
    END IF
NEXT i%
```

CALL

Type: Command (Control Flow)
Syntax:

- `CALL name`
- `CALL name arg1, arg2, ...`

Description: Calls a `SUB` procedure. Omit the argument list when the `SUB` has no parameters. The `CALL` keyword is optional when arguments are present. Parentheses are not permitted.
Example:

```
CALL DrawBox 10, 5, "*"
DrawBox 20, 3, "#"    ' Same as CALL DrawBox 20, 3, "#"
```

CASE

Type: Command (Control Flow)
Syntax: `CASE value`; `CASE value1, value2, ...`; `CASE value1# TO value2#`; `CASE IS = value#`, `CASE IS <> value#`, `CASE IS < value#`, `CASE IS > value#`, `CASE IS <= value#`, or `CASE IS >= value#`; or `CASE ELSE`
Description: Defines a case within a `SELECT CASE` statement.
Example:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
    CASE 80 TO 89
        PRINT "B"
    CASE ELSE
        PRINT "Other"
END SELECT
```

CATCH

Type: Command (Control Flow)
Syntax: `CATCH` or `CATCH errorVariable$`
Description: Catches errors raised in a `TRY` block. When `errorVariable$` is present, the error message (a string) is assigned to it. Used within `TRY … CATCH … FINALLY … END TRY` blocks.
Example:

```
TRY
    OPEN "data.txt" FOR TEXT READ AS file%
CATCH error$
    PRINT "Error: ", error$
END TRY
```

CIRCLE

Type: Command (Graphics)
Syntax: `CIRCLE AT (x%, y%) RADIUS radius%` or `CIRCLE AT (x%, y%) RADIUS radius%` ⟨`WITH paintExpr`⟩ ⟨`FILLED`⟩
Description: Convenience alias for `OVAL` when width equals height (a circle). **`WITH`** rules match **`RECTANGLE`** / **+OVAL**+. Coordinates use bottom-left origin (0,0).
Example:

```
COLOR &HFFFF00FF    ' Set global color to yellow
CIRCLE AT (320, 240) RADIUS 50
CIRCLE AT (100, 100) RADIUS 30 WITH &H00FF00FF FILLED
CIRCLE AT (200, 200) RADIUS 40 FILLED
```

CLAMP

Type: Command (Variables)
Syntax: `CLAMP variable FROM expression TO expression` or `CLAMP LOCAL variable FROM expression TO expression`
Description: Restricts a scalar integer or real variable to the inclusive interval between the two bounds (bounds may be given in either order). The stored type of the variable is preserved.
Example:

```
LET score% = 150
CLAMP score% FROM 0 TO 100
PRINT score%    ' 100
```

CLOSE

Type: Command (File I/O)
Syntax: `CLOSE fileHandle%`
Description: Closes an open file.
Example:

```
OPEN "data.txt" FOR TEXT READ AS file%
' ... read operations ...
CLOSE file%
```

CLS

Type: Command (Graphics)
Syntax: `CLS` or `CLS WITH backgroundColorExpr` (integers, variables, and CSS color name strings, same as `COLOR`)
Description: Clears the graphics screen to the current background color (default `&H000000FF` at run/reset). The IDE output uses a black mat behind the canvas where the buffer is transparent. Does not affect the text display. Resets the turtle so the next `TURTLE` begins at the center.
Example:

```
CLS
CLS WITH &H000033FF
CLS WITH "navy"
```

COLOR

Type: Command (Text/Graphics)
Syntax:

- `COLOR foregroundColor%`
- `COLOR foregroundColor% BACKGROUND backgroundColor%`
- `COLOR BACKGROUND backgroundColor%`

Description: Sets the global foreground and/or background color for both text and graphics operations. Colors are specified as 32-bit RGBA integers in `&HRRGGBBAA` format. These colors are used by default, but can be overridden in individual statements. The three syntax forms set foreground only, both colors, or background only (foreground unchanged), respectively. For text, both colors can apply. For graphics, only the foreground color is used for drawing (the `BACKGROUND` clause does not change pen color; use `WITH color%` per statement to override).
Example:

```
COLOR &HFF0000FF
PRINT "This is red text"
COLOR &HFFFFFF00 BACKGROUND &H000000FF
COLOR BACKGROUND &H000000FF
COLOR &H00FF00FF
LINE FROM (0, 0) TO (100, 100)
LINE FROM (50, 50) TO (150, 150) WITH &HFF0000FF
```

CONSOLE

Type: Statement (Console)
Syntax: `CONSOLE expression1, expression2, …` or `CONSOLE expression1; expression2; …` (same comma / semicolon rules as **+PRINT**, and at least one expression is required)
Description: Evaluates one or more expressions and prints **EBON-serialized** results to the **console** tab (not the graphics/text output). Useful for debugging. In program code you must use CONSOLE explicitly. Unlike PRINT, string values appear with EBON quoting (CONSOLE "5" → "5"; CONSOLE 5 → 5+).
Example:

```
CONSOLE 1 + 2
CONSOLE x%, y%
CONSOLE "5"          ' "5"
CONSOLE 5            ' 5
CONSOLE "Value: " + STR value#
```

Note: In the console input line, typing just an expression (e.g., `1 2`) automatically behaves like *CONSOLE+* with a single operand.

CONTINUE

Type: Command (Control Flow)
Syntax: `CONTINUE FOR` or `CONTINUE WHILE` or `CONTINUE DO` or `CONTINUE UNTIL`
Description: Skips the rest of the current loop iteration and continues with the next iteration. `CONTINUE UNTIL` applies to the innermost `UNTIL … UEND` block.
Example:

```
FOR i% = 1 TO 10
    IF i% MOD 2 = 0 THEN
        CONTINUE FOR    ' Skip even numbers
    END IF
    PRINT i%    ' Only prints odd numbers
NEXT i%
```

COPY

Type: Command (File I/O)
Syntax: `COPY "source" TO "destination"`
Description: Copies a file from source to destination.
Example:

```
COPY "data.txt" TO "backup.txt"
COPY "/source/file.bin" TO "/dest/file.bin"
```

DELETE

Type: Command (File I/O)
Syntax: `DELETE "filename"`
Description: Deletes a file.
Example:

```
DELETE "temp.txt"
DELETE "/Users/name/old_data.bin"
```

DIM

Type: Statement (Array Resizing)
Syntax:

- `DIM name[size]`
- `DIM name[rows, cols]`
- `DIM name[a, b, c, ...]`
- `DIM LOCAL name[size]`
- `DIM LOCAL name[rows, cols]`
- `DIM LOCAL name[a, b, c, ...]`

Description: Resizes an array using **per-axis lengths**; indices are **0-based**. Undeclared arrays are presumed to have size 0. `TO` ranges are **not** allowed in `DIM`. `DIM` is **only** for arrays; other variables do not need declaring. `LOCAL` binds the array to the current stack frame (see SUB and END SUB Statements).
Example:

```
DIM numbers%[10]
LET numbers%[] = [1, 2, 3]
DIM numbers%[20]
DIM matrix#[5, 10]            ' 5×10 matrix
```

DO

Type: Command (Control Flow)
Syntax:

- `DO ... LOOP` (no condition; use `EXIT DO` to break)
- `DO WHILE condition ... LOOP` (test before each iteration)
- `DO UNTIL condition ... LOOP` (test before each iteration)
- `DO ... LOOP WHILE condition` (test after each iteration)
- `DO ... LOOP UNTIL condition` (test after each iteration)

Description: Begins a `DO` loop. Top-tested forms skip the body when the condition fails on entry; bottom-tested forms always run the body at least once.
Example:

```
DO WHILE count% < 10
    PRINT count%
    LET count% += 1
LOOP

DO
    PRINT "Enter value: ";
    INPUT value%
LOOP UNTIL value% = 0
```

ELSE

Type: Command (Control Flow)
Syntax: `ELSE`
Description: Defines the alternative branch in an `IF` or `UNLESS` statement.
Example:

```
IF score% >= 60 THEN
    PRINT "Pass"
ELSE
    PRINT "Fail"
END IF
```

ELSEIF

Type: Command (Control Flow)
Syntax: `ELSEIF condition` with optional `THEN` when the clause ends the line; canonical form is `ELSEIF condition THEN`.
Description: Adds additional conditional branches to an `IF` statement. Use the single keyword **`ELSEIF`**; **`ELSE IF`** (two words) is **not** valid and will not parse.
Example:

```
IF age% < 13 THEN
    PRINT "Child"
ELSEIF age% < 20 THEN
    PRINT "Teenager"
ELSE
    PRINT "Adult"
END IF
```

END IF

Type: Command (Control Flow)
Syntax: `END IF`
Description: Terminates an `IF` statement block.
Example:

```
IF x% > 0 THEN
    PRINT "Positive"
END IF
```

END SELECT

Type: Command (Control Flow)
Syntax: `END SELECT`
Description: Terminates a `SELECT CASE` statement block.
Example:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
END SELECT
```

END SUB

Type: Command (Control Flow)
Syntax: `END SUB`
Description: Terminates a `SUB` procedure definition.
Example:

```
SUB PrintMessage (msg$)
    PRINT msg$
END SUB
```

END TRY

Type: Command (Control Flow)
Syntax: `END TRY`
Description: Terminates a `TRY ... CATCH ... FINALLY` block.
Example:

```
TRY
    OPEN "file.txt" FOR TEXT READ AS file%
CATCH error$
    PRINT "Error: ", error$
FINALLY
    CLOSE file%
END TRY
```

END UNLESS

Type: Command (Control Flow)
Syntax: `END UNLESS`
Description: Terminates an `UNLESS` statement block.
Example:

```
UNLESS valid%
    PRINT "Invalid input"
END UNLESS
```

EXIT

Type: Command (Control Flow)
Syntax: `EXIT` or `EXIT FOR` or `EXIT WHILE` or `EXIT DO` or `EXIT UNTIL` or `EXIT SUB`
Description: Bare `EXIT` terminates program execution immediately. Qualified forms exit the specified loop or `SUB` procedure. `EXIT UNTIL` breaks out of the innermost `UNTIL … UEND` block (jumps below `UEND`). For `DO … LOOP UNTIL`, use `EXIT DO`.
Example:

```
IF criticalError% THEN
    PRINT "Fatal error!"
    EXIT
END IF

FOR i% = 1 TO 100
    IF found% THEN
        EXIT FOR
    END IF
NEXT i%

DO
    IF quit% THEN
        EXIT DO
    END IF
LOOP
```

FINALLY

Type: Command (Control Flow)
Syntax: `FINALLY`
Description: Marks the cleanup section of a `TRY … CATCH … FINALLY … END TRY` block. The `FINALLY` block always executes, whether an error occurred or not. Optional; you can use `TRY … CATCH … END TRY` without `FINALLY`.
Example:

```
TRY
    OPEN "data.txt" FOR TEXT READ AS file%
    READFILE content$ FROM "data.txt"
CATCH error$
    PRINT "Error: ", error$
FINALLY
    CLOSE file%
END TRY
```

FOR

Type: Command (Control Flow)
Syntax:

- `FOR variable = start TO end ⟨STEP step⟩`
- `FOR LOCAL variable = start TO end ⟨STEP step⟩`

Description: Begins a `FOR` loop that iterates a variable through a range. `LOCAL` creates the counter in the current stack frame. `STEP` defaults to `1` when omitted; a negative `STEP` counts down.
Example:

```
FOR i% = 1 TO 10
    PRINT i%
NEXT i%

FOR x# = 10 TO 0 STEP -0.5
    PRINT x#
NEXT x#
```

GET

Type: Command (Graphics)
Syntax:

- `GET spriteArray[,] FROM (x1, y1) TO (x2, y2)`
- `GET LOCAL spriteArray[,] FROM (x1, y1) TO (x2, y2)`

Description: Captures a rectangular region of the screen into a **two-dimensional integer array** (sprite matrix). `LOCAL` binds the sprite array to the current stack frame. Replaces the array with **`EXTENTS sprite%[,] = [height, width]`**; each cell is **0xRRGGBBAA at sprite%[row, col] (`row` 0+ is the bottom row). Coordinates use bottom-left origin (0,0). The same matrix can be passed to `WITH` on `PAINT` or `FILLED RECTANGLE/OVAL/CIRCLE/TRIANGLE` for a tiled pattern (<<interior-paint,**Interior paint>> under Graphics).**
Example:*

```
GET sprite%[,] FROM (100, 100) TO (131, 131)
```

GOTO

Type: Command (Control Flow)
Syntax: `GOTO labelName`
Description: Transfers control unconditionally to the specified label.
Example:

```
LET x% = 0
LABEL Loop
    PRINT x%
    LET x% += 1
    IF x% < 5 THEN
        GOTO Loop
    END IF
```

HELP

Type: Statement (Console)
Syntax: `HELP topic`
Description: Displays every registered syntax line for a statement-start keyword, a named expression operator (for example `CHR`, `MOD`), or a punctuation operator (`` ``, `=`, `<>`, `^`, `,` etc., `not` . `or` (+)). Prints one line per form in the **console** tab (not the graphics/text output). If nothing is registered for that topic, prints a short not-found message.
Example:

```
HELP PRINT
HELP COLOR
HELP FOR
HELP +
HELP **
```

Output: Each valid syntax form for the statement is printed on a separate line in the console.

IF

Type: Command (Control Flow)
Syntax:

- `IF condition ⟨THEN⟩ ... END IF`
- `IF condition ⟨THEN⟩ ... ⟨ELSEIF condition ⟨THEN⟩ ...⟩ ⟨ELSE ...⟩ END IF`

Description: Executes code conditionally based on a boolean expression. `THEN` may be omitted when the header ends the line. `ELSEIF` and `ELSE` add additional branches before `END IF`.
Example:

```
IF score% >= 90 THEN
    PRINT "A"
END IF

IF temperature# > 100 THEN
    PRINT "Boiling!"
END IF
```

INPUT

Type: Command (Text I/O)
Syntax:

- `INPUT variable`
- `INPUT LOCAL variable`

Description: Reads a value from the user and assigns it to a variable or array target. The variable's type sigil determines what type of value is expected. `LOCAL` creates or writes a stack-frame-local target. The prompt is displayed separately using `PRINT` before the `INPUT` statement. For arrays, reads multiple comma-separated values, filling the array from index **0** onwards.
Example:

```
PRINT "Enter your age: ";
INPUT age%
PRINT "Enter your name: ";
INPUT name$

DIM scores%[5]
PRINT "Enter 5 scores (comma-separated): ";
INPUT scores%[]
```

INSERT

Type: Statement (Array)
Syntax: `INSERT value AT index INTO array[]`
Description: Inserts `value` before the element at 0-based `index` in a 1D array. Valid indices are **0** through the current length (inclusive). Index **0** prepends; index equal to length appends. Index greater than length is a runtime error.
Example:

```
INSERT 9 AT 1 INTO numbers%[]
INSERT "Alice" AT 0 INTO names$[]
```

LABEL

Type: Command (Control Flow)
Syntax: `LABEL labelName`
Description: Defines a label that can be targeted by `GOTO`.
Example:

```
LABEL StartProgram
PRINT "Starting..."

LABEL MainLoop
PRINT "Looping..."
```

LET

Type: Statement (Variable Assignment)
Syntax:

- `LET variable = expr`
- `LET variable` = expr · LET variable -= expr · LET variable *= expr · LET variable /= expr · LET variable ^= expr+
- `LET array[] = expr`
- `LET array[index] = expr`
- `LET array[index1, index2, ...] = expr`
- `LET structure.member = expr`

Description: Assigns a value to a module-level (global) target. Compound tokens (`=, -=+, etc.) are shorthand using ordinary arithmetic operators. Array targets accept literals, another array, or a slice expression. Member and index paths may be combined (see LET and LOCAL statements).
Example:

```
LET count% = 10
LET name$ = "Alice"
LET result# = 3.14159
LET complex& = 3+4i

' Array assignments
LET nums%[] = [10, 20, 30, 40, 50]
```

```
LET copy%[] = nums%[]
LET slice%[] = nums%[2 TO 4]    ' [30, 40, 50]
```

LINE

Type: Command (Graphics)
Syntax: `LINE FROM (x1%, y1%) TO (x2%, y2%)` or `LINE FROM (x1%, y1%) TO (x2%, y2%) WITH color%`
Description: Draws a line between two points. Uses global foreground color by default, or `WITH color%` to override. For filled rectangles, use `RECTANGLE … FILLED`. Coordinates use bottom-left origin (0,0).
Example:

```
LINE FROM (10, 10) TO (100, 50) WITH &H00FF00FF
LINE FROM (200, 200) TO (300, 300) WITH &HFF0000FF
```

LISTDIR

Type: Command (File I/O)
Syntax:

- `LISTDIR array[] FROM path`
- `LISTDIR LOCAL array[] FROM path`

Description: Lists files in a directory and stores filenames in an array. `LOCAL` binds the result array to the current stack frame. Indices are **0-based**; entries include files and subdirectories.
Example:

```
LISTDIR files$[] FROM "."
FOR i% = 0 TO LEN files$[] - 1
    PRINT files$[i%]
NEXT i%
```

LOCAL

Type: Statement (Variable Assignment)
Syntax:

- `LOCAL variable = expr`
- `LOCAL variable` = expr · LOCAL variable -= expr · LOCAL variable *= expr · LOCAL variable /= expr · LOCAL variable ^= expr+
- `LOCAL array[] = expr`

Description: Creates a variable local to the current stack frame. Inside a `SUB`, the variable is destroyed when the `SUB` exits; at top level it is scoped to the main program frame. Compound assignment uses the same shorthand rules as `LET`. Many other statements also accept `LOCAL` before a target name (`FOR`, `INPUT`, `READ`, `OPEN … AS`, `DIM`, `CLAMP`, `SWAP`, and others; see LET and LOCAL statements).
Example:

```
SUB Calculate ()
    LOCAL temp% = 10    ' Local to this SUB
    LOCAL result# = 0   ' Local to this SUB
    LOCAL buffer%[] = [0, 0, 0]    ' Local array
END SUB
```

LOCATE

Type: Command (Text I/O)
Syntax: `LOCATE ROW row%` · `LOCATE COLUMN column%` · `LOCATE ROW row% COLUMN column%` (at least one of `ROW` or `COLUMN`; when both are present, canonical spelling is `ROW` then `COLUMN`)
Description: Positions the text cursor at a specific row and column in the text display. Row and column are 0-based (row 0, column 0 is the top-left corner). Valid column range: **0–79**. Valid row range: **0–29** with line spacing off (80×30 grid), or **0–23** with line spacing on (80×24 grid). A **wide** Unifont glyph printed after `LOCATE` occupies two columns starting at `column%`, so use column **78** or lower if the next printed character is wide.
Example:

```
LOCATE ROW 9 COLUMN 19
PRINT "This text appears at row 9, column 19"
```

LOOP

Type: Command (Control Flow)
Syntax: `LOOP` or `LOOP WHILE condition` or `LOOP UNTIL condition`
Description: Marks the end of a `DO` loop, with optional condition at bottom.
Example:

```
DO
    PRINT "Hello"
LOOP WHILE continue%

DO
    PRINT "Value: ";
    INPUT val%
LOOP UNTIL val% = 0
```

MKDIR

Type: Command (File I/O)
Syntax: `MKDIR "path"`
Description: Creates a directory.
Example:

```
MKDIR "backups"
MKDIR "/Users/name/data"
```

MOVE

Type: Command (File I/O)
Syntax: `MOVE "source" TO "destination"`
Description: Moves or renames a file.
Example:

```
MOVE "oldname.txt" TO "newname.txt"
MOVE "/temp/file.txt" TO "/final/file.txt"
```

NEXT

Type: Command (Control Flow)
Syntax:

- `NEXT`
- `NEXT variable`
- `NEXT LOCAL variable`

Description: Marks the end of a `FOR` loop. The variable name is optional when only one `FOR` loop is active. `NEXT LOCAL variable` must match a `FOR LOCAL` counter in the same frame.
Example:

```
FOR i% = 1 TO 10
    PRINT i%
NEXT i%

FOR j% = 1 TO 5
    PRINT j%
NEXT
```

OPEN

Type: Command (File I/O)
Syntax:

- `OPEN filename FOR TEXT READ AS handle`
- `OPEN filename FOR TEXT APPEND AS handle`
- `OPEN filename FOR TEXT OVERWRITE AS handle`
- `OPEN filename FOR DATA READ AS handle`
- `OPEN filename FOR DATA OVERWRITE AS handle`
- `OPEN filename FOR TEXT READ AS LOCAL handle`
- `OPEN filename FOR TEXT APPEND AS LOCAL handle`
- `OPEN filename FOR TEXT OVERWRITE AS LOCAL handle`
- `OPEN filename FOR DATA READ AS LOCAL handle`
- `OPEN filename FOR DATA OVERWRITE AS LOCAL handle`

Description: Opens a file and assigns a handle. You must specify **both** a **record format** (`TEXT` or `DATA`) and an **access mode** (`READ`, `APPEND`, or `OVERWRITE`) after `FOR`. `LOCAL` binds the handle variable to the current stack frame.

- **`TEXT`**: Subsequent `READ`/`WRITE` use plain UTF-8 **lines** (strings only on `WRITE`; string variables on `READ`). `SEEK`, `LOC`, and `EOF` use a byte cursor.
- **`DATA`**: One EBON **document** per file: `READ` loads the entire file into one variable; each `WRITE` replaces the entire file with one value. `APPEND` is not supported. `SEEK` and `LOC` raise an error; `EOF` is supported.

Access mode (`READ`, `APPEND`, or `OVERWRITE`) controls read-only vs writing. `APPEND` applies to `TEXT` only. Attempting to open a non-existent file in `READ` causes an error; `OVERWRITE` (and `TEXT APPEND`) create the file if needed.
Example:

```
OPEN "data.txt" FOR TEXT READ AS inputFile%
OPEN "output.txt" FOR TEXT OVERWRITE AS outputFile%
OPEN "log.txt" FOR TEXT APPEND AS logFile%
OPEN "snapshot.ebon" FOR DATA OVERWRITE AS dataOut%
```

OVAL

Type: Command (Graphics)
Syntax: `OVAL AT (x%, y%) RADII (radiusX%, radiusY%)` or `OVAL AT (x%, y%) RADII (radiusX%, radiusY%) ⟨WITH paintExpr⟩ ⟨FILLED⟩`
Description: Draws an oval (ellipse) outline or filled oval. **`WITH`** rules match **`RECTANGLE`** (solid or sprite tile when **+FILLED**). `A circle is a special case where` radiusX% `equals` radiusY%+. Coordinates use bottom-left origin (0,0).
Example:

```
COLOR &HFFFF00FF    ' Set global color to yellow
OVAL AT (320, 240) RADII (100, 100)
OVAL AT (100, 100) RADII (60, 60) WITH &H00FF00FF FILLED
OVAL AT (200, 200) RADII (80, 40) WITH &HFF00FFFF FILLED
```

PAINT

Type: Command (Graphics)
Syntax: `PAINT (x%, y%) ⟨WITH paintExpr⟩`
Description: Flood-fills a bounded region starting at (x%, y%). **`WITH`** may be a solid RGBA integer, a CSS color **string**, or a **+GET**+-format **sprite array** tiled with phase at screen origin (0,0). Coordinates use bottom-left origin (0,0).
Example:

```
PAINT (100, 100) WITH &HFF0000FF    ' Fill area with red
PAINT (200, 200) WITH &H00FF00FF    ' Fill with green
GET pat%[,] FROM (0, 0) TO (3, 3)
PAINT (150, 150) WITH pat%[,]    ' Tiled pattern fill
```

PLAY

Type: Command (Audio)
Syntax: `PLAY voice%, mml$`
Description: Plays music or sound on a specific voice using Music Macro Language (MML). **All PLAY statements play music in the background**, execution continues immediately without waiting for the music to finish. `voice%` identifies the voice (**1–16**). `mml$` is the MML string; full rules are under **MML Syntax Reference** in Audio, with the same tokens summarized in Appendix: MML Syntax. The voice's timbre comes from `VOICE voice INSTRUMENT instrument`. Multiple voices can play simultaneously. Each voice has a queue of up to 65,536 entries (each MML token counts as one); if `PLAY` would exceed that, a **Music Overflow** error is raised. Use **NOTES** to check how many entries remain. MML **V** is relative to global **+VOLUME+**.
Example:

```
VOICE 1 INSTRUMENT 1
TEMPO 120
PLAY 1, "CDEFGAB C"
PLAY 1, "V100 C L4 V64 D L4"
PLAY 1, "N60 N64 N67 L2"

' Check how many notes remain
LET notesLeft% = NOTES 1
IF notesLeft% > 0 THEN
    PRINT "Still playing: ", notesLeft%, " notes remaining"
END IF
```

PLUCK

Type: Statement (Array)
Syntax:

- `PLUCK value FROM array[]`
- `PLUCK value FROM array[] INTO variable`
- `PLUCK value FROM array[] INTO LOCAL variable`
- `PLUCK ALL value FROM array[]`
- `PLUCK AT index FROM array[]`
- `PLUCK AT index FROM array[] INTO variable`
- `PLUCK AT index FROM array[] INTO LOCAL variable`

Description: Removes element(s) from a 1D array. **By value:** removes the **first** match (`valuesEqual` semantics). **By index:** removes at 0-based `index`. **All matches:** `PLUCK ALL` removes every match (high to low); no `INTO`; zero matches is a no-op. Optional `INTO` assigns the removed element; `LOCAL` binds the `INTO` target to the current frame. Omit `INTO` to discard. Value not found (non-`ALL` by-value form) is a runtime error.
Example:

```
PLUCK "hawthorn" FROM inventory$[]
PLUCK 5 FROM numbers%[] INTO value%
PLUCK AT 2 FROM names$[]
PLUCK ALL "x" FROM tags$[]
```

POP

Type: Statement (Array)
Syntax:

- `POP array[]`
- `POP array[] INTO variable`
- `POP array[] INTO LOCAL variable`

Description: Removes the last element from an array. Omit `INTO` to discard the element; `INTO` assigns it to a variable; `LOCAL` binds the `INTO` target to the current stack frame.
Example:

```
POP numbers%[]
POP numbers%[] INTO value%
POP names$[] INTO name$
```

PRINT

Type: Command (Text I/O)
Syntax: `PRINT expression1, expression2, ...⟨;⟩ ⟨OPAQUE⟩` or `PRINT array[]⟨;⟩ ⟨OPAQUE⟩` or `PRINT ⟨OPAQUE⟩`
Description: Outputs text and values to the text display. The `PRINT` statement accepts expressions of any type (Integer, Real, Complex, String, Array) separated by commas. Each expression is converted to its string representation. **Comma (`,`)** inserts a single space before the next value; **semicolon (`;`)** inserts no extra space. Comma or semicolon after the last item suppresses the trailing newline. Empty `PRINT` outputs a blank line. When printing an array, all elements are printed in brackets with spaces after commas (e.g., `[1, 2, 3, 4, 5]`). **OPAQUE** replaces text cell pixels instead of alpha-compositing over existing graphics.
Example:

```
PRINT "Hello, world!"
PRINT "Name: ", name$, " Age: ", age%
PRINT "X: ", x%, " Y: ", y%

' Mixed types
PRINT name$, count%, price#, z&

PRINT numbers%[]
PRINT scores%[]
PRINT "Enter name: ";
PRINT

PRINT "Name: " + name$ + " Age: " + STR age%
```

PSET

Type: Command (Graphics)
Syntax: `PSET (x%, y%) ⟨WITH color%⟩ ⟨OPAQUE⟩`
Description: Sets a single pixel. With `WITH color%`, uses that color; otherwise uses the global foreground color from `COLOR`. **OPAQUE** replaces the destination pixel instead of alpha-compositing. Coordinates use bottom-left origin (0,0). X ranges from 0 to 639, Y ranges from 0 to 479. Color is a 32-bit RGBA integer.
Example:

```
PSET (100, 200) WITH &HFF0000FF    ' Red pixel
FOR i% = 0 TO 639
```

```
    PSET (i%, 240) WITH &HFFFFFFFF
NEXT i%
```

PUSH

Type: Statement (Array)
Syntax: `PUSH value INTO array[]`
Description: Adds an element to the end of an array.
Example:

```
PUSH 10 INTO numbers%[]
PUSH "David" INTO names$[]
```

PUT

Type: Command (Graphics)
Syntax: `PUT spriteArray%[,] AT (x%, y%) ⟨OPAQUE⟩`
Description: Draws a sprite from a **GET** 2D integer matrix onto the screen. The sprite's bottom-left corner is positioned at (x%, y%). Pixels are alpha-blended using the alpha channel from the **0xRRGGBBAA** values. Transparent pixels (alpha = 0) are not drawn. **OPAQUE** writes every sprite pixel directly, including transparent pixels. The same matrix can be used as **WITH** on **PAINT** or **FILLED** shapes for tiled fills. Coordinates use bottom-left origin (0,0).
Example:

```
PUT sprite%[,] AT (200, 150)
PUT player%[,] AT (x%, y%)    ' Animate sprite
```

RANDOMIZE

Type: Command (Random)
Syntax: `RANDOMIZE` or `RANDOMIZE seed%`
Description: Seeds the random number generator. Without argument, uses `NOW%`.
Example:

```
RANDOMIZE               ' Seed with current timestamp (NOW%)
RANDOMIZE 12345         ' Seed with specific value
RANDOMIZE NOW%
```

READ

Type: Command (File I/O)
Syntax:

- `READ ⟨LOCAL⟩ variable⟨, ⟨LOCAL⟩ variable ...⟩ FROM handle`
- `READ ⟨LOCAL⟩ array[] FROM handle`

Description: Reads from an open file. `TEXT` mode: one line per scalar variable (string variables only). `DATA` mode: exactly one variable or array; loads the entire file as one EBON document. Optional `LOCAL` on each target binds that variable to the current stack frame. A second `DATA READ` errors after the document is consumed (check `EOF`).
Example:

```
OPEN "game.ebon" FOR DATA READ AS file%
READ gameConfig FROM file%
CLOSE file%

OPEN "data.txt" FOR TEXT READ AS file%
WHILE NOT EOF file%
    READ line$ FROM file%
    PRINT line$
WEND
CLOSE file%
```

READFILE

Type: Command (File I/O)
Syntax:

- `READFILE variable FROM filename`
- `READFILE LOCAL variable FROM filename`

Description: Reads an entire file into a string variable. `LOCAL` binds the target to the current stack frame.
Example:

```
READFILE config$ FROM "config.txt"
PRINT config$
READFILE jsonData$ FROM "data.json"
```

RECTANGLE

Type: Command (Graphics)
Syntax: `RECTANGLE FROM (x1%, y1%) TO (x2%, y2%)` or `RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) ⟨WITH paintExpr⟩ ⟨FILLED⟩`
Description: Draws a rectangle outline or filled rectangle. Uses global foreground color by default, or **`WITH`** to override. With **+FILLED**, ⋆WITH **may be solid RGBA, a CSS color string, or a** GET**+-format sprite array (tiled at origin (0,0)); outline-only** `WITH` **must be solid. Default is outline only. Coordinates use bottom-left origin (0,0).**
Example:*

```
COLOR &HFFFFFFFF
RECTANGLE FROM (50, 50) TO (150, 150)
RECTANGLE FROM (200,200) TO (300,300) WITH &HFF0000FF FILLED
```

REVERSE

Type: Statement (Array)
Syntax: `REVERSE array[]`
Description: Reverses the element order of a 1D array **in place**.
Example:

```
REVERSE numbers%[]
REVERSE names$[]
```

RMDIR

Type: Command (File I/O)
Syntax: `RMDIR "path"`
Description: Removes an empty directory.
Example:

```
RMDIR "temp"
RMDIR "/Users/name/old_data"
```

SEEK

Type: Command (File I/O)
Syntax: `SEEK position% IN fileHandle%`
Description: Positions the file pointer at a specific byte position (0-based) on a `TEXT` handle. Not supported in `DATA` mode. Position 0 is the beginning of the file. Seeking past end of file is allowed on writable `TEXT` handles (file will extend on write).
Example:

```
OPEN "log.txt" FOR TEXT READ AS file%
SEEK 0 IN file%
READ line$ FROM file%
CLOSE file%
```

SELECT CASE

Type: Command (Control Flow)
Syntax: `SELECT CASE expression ... END SELECT`
Description: Multi-way branching based on the value of an expression.
Example:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
    CASE 80 TO 89
        PRINT "B"
    CASE ELSE
        PRINT "Other"
END SELECT
```

SET

Type: Command (System Settings)
Syntax: `SET LINE SPACING ON` / `SET LINE SPACING OFF` / `SET TEXT WRAP ON` / `SET TEXT WRAP OFF` / `SET AUDIO ON` / `SET AUDIO OFF` / `SET FULLSCREEN ON` / `SET FULLSCREEN OFF` / `SET SCROLL ON` / `SET SCROLL OFF`
Description: Configures the IDE and runtime. Full coverage: SET LINE SPACING Statement, SET TEXT WRAP Statement, SET SCROLL Statement, and SET FULLSCREEN Statement under Console and Text I/O; SET AUDIO Statement under Audio.
Example:

```
SET LINE SPACING OFF
SET LINE SPACING ON

SET TEXT WRAP OFF
SET TEXT WRAP ON

SET AUDIO ON
SET AUDIO OFF

SET FULLSCREEN ON
SET FULLSCREEN OFF

SET SCROLL ON
SET SCROLL OFF
```

SHIFT

Type: Statement (Array)
Syntax:

- `SHIFT array[]`
- `SHIFT array[] INTO variable`
- `SHIFT array[] INTO LOCAL variable`

Description: Removes the first element from an array. Omit `INTO` to discard the element; `INTO` assigns it to a variable; `LOCAL` binds the `INTO` target to the current stack frame.
Example:

```
SHIFT numbers%[]
SHIFT numbers%[] INTO value%
SHIFT names$[] INTO name$
```

SLEEP

Type: Command (Control Flow)
Syntax: `SLEEP milliseconds%`
Description: Pauses program execution for the specified number of milliseconds. `milliseconds%` must be a non-negative integer. Keyboard input and background audio may still be processed during the pause.
Example:

```
PRINT "Starting in 3 seconds..."
SLEEP 1000
PRINT "Go!"
```

STEP

Type: Keyword (Control Flow)
Syntax: Used within `FOR` loops
Description: Specifies the increment/decrement value for a `FOR` loop.
Example:

```
FOR i% = 0 TO 100 STEP 10
    PRINT i%
NEXT i%
```

SUB

Type: Command (Control Flow)
Syntax:

- `SUB name param1, param2 ... END SUB`
- `SUB name BYREF param1, param2 ... END SUB`

Description: Defines a subroutine procedure. `BYREF` passes parameters by reference so the `SUB` can modify caller variables. Parentheses are not permitted. Use `CALL name` or `CALL name args` to invoke; `EXIT SUB` returns early.
Example:

```
SUB DrawBox width%, height%, char$
    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB
```

SWAP

Type: Command (Variables)
Syntax: `SWAP target1, target2` or `SWAP LOCAL target1, LOCAL target2` (optional `LOCAL` on each operand)
Description: Exchanges the values at two assignment targets. Each target may use any valid `LET` / `LOCAL` left-hand form (scalars, members, indices, chained paths). Operand values must have the same type; array operands must share the same element type.
Example:

```
LET a% = 1
LET b% = 2
SWAP a%, b%
PRINT a%, b%    ' 2  1

DIM arr%[3]
LET arr%[0] = 10
LET arr%[2] = 30
SWAP arr%[0], arr%[2]
```

TEMPO

Type: Command (Audio)
Syntax: `TEMPO beatsPerMinute%`
Description: Sets the playback tempo for MML music in beats per minute (BPM). Affects all subsequent `PLAY` statements using MML. Default tempo is typically 120 BPM.
Example:

```
TEMPO 120    ' Set to 120 BPM (moderate tempo)
TEMPO 60     ' Set to 60 BPM (slow)
TEMPO 180    ' Set to 180 BPM (fast)
```

THEN

Type: Keyword (Control Flow)
Syntax: Used with `IF` and `UNLESS`
Description: Separates the condition from the branch body in conditional statements (the body starts on the following lines).
Example:

```
IF x% > 0 THEN
    PRINT "Positive"
END IF
```

THROW

Type: Command (Control Flow)
Syntax: `THROW errorMessage$`
Description: Throws an error. The error message is a string. Control immediately transfers to the nearest `CATCH` block, or the program terminates if no `CATCH` block exists.
Example:

```
IF divisor% = 0 THEN
    THROW "Division by zero"
END IF

IF NOT EXISTS filename$ THEN
    THROW "File not found: " + filename$
END IF
```

TO

Type: Keyword (Control Flow)
Syntax: Used within `FOR` loops and `CASE` clauses
Description: Specifies a range in `FOR` loops or `CASE` statements.
Example:

```
FOR i% = 1 TO 10
NEXT i%

CASE 90 TO 100
    PRINT "A"
```

TRIANGLE

Type: Command (Graphics)
Syntax: `TRIANGLE (x1%, y1%), (x2%, y2%), (x3%, y3%)` or `TRIANGLE (x1%, y1%), (x2%, y2%), (x3%, y3%) ⟨WITH paintExpr⟩ ⟨FILLED⟩`
Description: Draws a triangle with vertices at the three specified points. With **+FILLED**, ⋆WITH **may be solid or a** GET**+-format sprite tile (same as other** `FILLED` **shapes). Outline-only** `WITH` **must be solid. Coordinates use bottom-left origin (0,0).**
Example:*

```
TRIANGLE (100, 100), (200, 200), (150, 250) WITH &HFFFFFFFF
TRIANGLE (50,50), (150,50), (100,150) WITH &HFF0000FF FILLED
```

TRY

Type: Command (Control Flow)
Syntax: `TRY ... CATCH ⟨errorVariable$⟩ ... FINALLY ... END TRY`
Description: Begins a structured error handling block. The `TRY` block contains code that might raise an error. Errors are represented as strings. `CATCH` and `FINALLY` are optional.
Example:

```
TRY
    OPEN "data.txt" FOR TEXT READ AS file%
    READFILE content$ FROM "data.txt"
CATCH error$
    PRINT "Error: ", error$
FINALLY
    CLOSE file%
END TRY
```

TURTLE

Type: Command (Graphics)
Syntax: `TURTLE command$`
Description: Executes Logo-style turtle graphics on the 640×480 canvas using the current foreground color. The turtle's position, angle, and pen state persist until `CLS` or execution reset. Commands: `FD n` (forward), `BK n` (back), `RT n` (right turn), `LT n` (left turn), `PU` (pen up), `PD` (pen down), `HOME` (return to center; **case-insensitive**). Full table: Appendix: Turtle Commands.
Example:

```
TURTLE "FD 100 RT 90 FD 100 RT 90 FD 100 RT 90 FD 100"
TURTLE "FD 100 RT 120 FD 100 RT 120 FD 100"
```

UEND

Type: Command (Control Flow)
Syntax: `UEND`
Description: Marks the end of an `UNTIL` loop.
Example:

```
UNTIL count% >= 10
    PRINT count%
    LET count% += 1
UEND
```

UNLESS

Type: Command (Control Flow)
Syntax: `UNLESS condition` … `END UNLESS`, optionally with `ELSE` … before `END UNLESS`; `THEN` may be omitted when the header ends the line (canonical text never includes `THEN`)
Description: Syntactic sugar for `IF NOT`. Executes when condition is false.
Example:

```
UNLESS password$ = "secret"
    PRINT "Access denied"
END UNLESS

UNLESS balance# >= price#
    PRINT "Insufficient funds"
ELSE
    LET balance# -= price#
    PRINT "Purchase complete"
END UNLESS
```

UNSHIFT

Type: Statement (Array)
Syntax: `UNSHIFT value INTO array[]`
Description: Adds an element to the beginning of an array.
Example:

```
UNSHIFT 0 INTO numbers%[]
UNSHIFT "Alice" INTO names$[]
```

UNTIL

Type: Command (Control Flow)
Syntax: `UNTIL condition ... UEND`
Description: Syntactic sugar for `WHILE NOT`. Repeats a block until a condition becomes true. Condition tested before each iteration.
Example:

```
LET count% = 0
UNTIL count% >= 10
    PRINT count%
    LET count% += 1
UEND

LET input$ = ""
UNTIL input$ = "quit"
    PRINT "Enter command (or 'quit'): ";
    INPUT input$
    PRINT "You entered: ", input$
UEND
```

VOICE

Type: Command (Audio)
Syntax:

```
VOICE voice% INSTRUMENT instrumentNumber%
VOICE voice% INSTRUMENT instrumentName$
```

Description: Assigns a General MIDI instrument to a voice. The instrument sets the timbre when the voice plays MML via `PLAY`. `voice%` identifies the voice (**1–16**). **Instrument by number:** `INSTRUMENT instrumentNumber%` uses a **melodic** GM program number (**1–128**) on **bank 0**, e.g. **1** = Acoustic Grand Piano, **41** = Violin. **Instrument by name:** `INSTRUMENT instrumentName$` matches a **drum-kit** preset first if possible (e.g. `"Standard 1 Kit"`; see Appendix: Drum Kits), otherwise a **melodic** name (**bank 0** first, then auxiliary banks in the SoundFont). Names are case-insensitive. Voices retain their instrument until changed by another `VOICE` statement.
Example:

```
VOICE 1 INSTRUMENT 1    ' Acoustic Grand Piano
VOICE 2 INSTRUMENT 28  ' Electric Guitar (clean)
VOICE 2 INSTRUMENT "Violin"
PLAY 1, "CDEFGAB C"
PLAY 2, "CEG"
```

VOLUME

Type: Command (Audio)
Syntax: `VOLUME volumeLevel%`
Description: Sets the global volume for all audio output. Volume level is **0–100**, where 0 = silent and 100 = maximum volume. Default volume is **100**. Affects all voices regardless of their configuration. Individual voice velocity (V in MML) is relative to the global volume.
Example:

```
VOLUME 100    ' Maximum volume (default)
VOLUME 50     ' Half volume
VOLUME 25     ' Quarter volume
VOLUME 0      ' Silent
```

WAIT

Type: Command (Control Flow)
Syntax: `WAIT WHILE condition` or `WAIT UNTIL condition`
Description: Single-line busy wait. Re-evaluates an integer condition on every interpreter step and stays on that source line until the wait ends. `WAIT WHILE` waits while the condition is true (like an empty `WHILE … WEND`). `WAIT UNTIL` waits until the condition is true (like an empty `UNTIL … UEND`). Use for short polls; avoid tight infinite waits. See **WAIT WHILE and WAIT UNTIL Statements** under Control Flow.
Example:

```
WAIT WHILE NOTES 1 > 0

LET ready% = 0
WAIT UNTIL ready%
PRINT "Ready."
```

WEND

Type: Command (Control Flow)
Syntax: `WEND`
Description: Marks the end of a `WHILE` loop.
Example:

```
WHILE count% < 10
    PRINT count%
    LET count% += 1
WEND
```

WHILE

Type: Command (Control Flow)
Syntax: `WHILE condition ... WEND`
Description: Repeats a block while a condition is true. Condition tested before each iteration.
Example:

```
LET count% = 0
WHILE count% < 10
    PRINT count%
    LET count% += 1
WEND
```

WRITE

Type: Command (File I/O)
Syntax: `WRITE expression⟨, expression ...⟩ TO fileHandle%`
Description: Writes to an open file. `TEXT` mode: one line per expression (strings only). `DATA` mode: exactly one expression; each `WRITE` replaces the entire file with one pretty-printed EBON document plus trailing newline.
Example:

```
OPEN "output.txt" FOR TEXT OVERWRITE AS file%
WRITE "Name: ", playerName$ TO file%
CLOSE file%

OPEN "save.ebon" FOR DATA OVERWRITE AS file%
WRITE gameConfig TO file%
CLOSE file%
```

WRITEFILE

Type: Command (File I/O)
Syntax: `WRITEFILE expression TO filename$`
Description: Writes an entire string to a file.
Example:

```
LET report$ = "Sales Report" + CHR 10 + "Total: $1000"
WRITEFILE report$ TO "report.txt"
WRITEFILE output$ TO "results.txt"
```

Appendix: Operator Reference

This appendix is an alphabetical reference of **EduBASIC expression operators** and related computational forms (`=== NAME` entries). It does not list **statements** or **commands**; those are in Appendix: Statement Reference.

Overview tables and tutorial prose for operators appear in the main guide, especially Numerical Operations, Strings (String Operators, STR, VAL, HEX, and BIN, String Templates), ch-arrays (Array Operators), Console and Text I/O for `PRINT` and `INPUT`, File I/O (`EOF`, `LOC`, `EXISTS`), and Audio (`NOTES`). Parentheses, literals, subscripts, and member access are covered under Data Types, ch-arrays, and ch-structures.

Operator Precedence Summary

This appendix is the authoritative precedence reference for EduBASIC expression operators.

EduBASIC follows standard mathematical operator precedence:

1. **Nullary Operators**: `RND#`, `INKEY$[]`, `PI#`, `E#`, `DATE$`, `TIME$`, `NOW%`, `TRUE%`, `FALSE%`, `CRSRLIN%`, `CRSRCOL%` (highest precedence)
2. Parentheses ( )
3. **Prefix operators**: `SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ATAN`, `SINH`, `COSH`, `TANH`, `ASINH`, `ACOSH`, `ATANH`, `EXP`, `LOG`, `LOG10`, `LOG2`, `SQRT`, `CBRT`, `FLOOR`, `CEIL`, `ROUND`, `TRUNC`, `EXPAND`, `SGN`, `ABS`, `REAL`, `IMAG`, `CONJ`, `CARG`, `INT`, `ASC`, `CHR`, `STR`, `VAL`, `HEX`, `BIN`, `UCASE`, `LCASE`, `LTRIM`, `RTRIM`, `TRIM`, `LEN`, `EXTENTS`, `EOF`, `LOC`, `EXISTS`, `NOTES`
4. **Postfix operators**: `!` (factorial), `DEG`, `RAD`
5. Unary and -
6. Exponentiation ^ or (right-associative)
7. Multiplication `*`, Division `/`, and Modulo `MOD`
8. Addition and Subtraction +- (also array concatenation)
9. Array search operators: `INDEXOF`, `LASTINDEXOF`, `INCLUDES`, `CONTAINS`
10. String operators: `INSTR`, `JOIN`, `REPLACE`, `LEFT`, `RIGHT`, `MID`, `STARTSWITH`, `ENDSWITH`
11. Comparison operators (=, <>, <, **>**, <=, >=)
12. `NOT` (unary logical)

13. **Logical AND**: `AND`, `NAND`
14. **Logical OR**: `OR`, `NOR`
15. **Logical XOR**: `XOR`, `XNOR`
16. **Logical IMP**: `IMP` (lowest precedence)

When operators have the same precedence, evaluation proceeds left to right, except for exponentiation which is right-associative.

! (Factorial)

Type: Operator (Arithmetic)
Syntax: `expr!`
Description: Postfix factorial. Operand must be a non-negative integer; returns the product `1 × 2 × … × n`.
Example:

```
LET factorialNum% = 5!    ' 120
```

** (Multiplication)*

Type: Operator (Arithmetic)
Syntax: `expr1 * expr2`
Description: Multiplies two numeric operands.
Example:

```
LET product% = 6 * 7
```

+ (Addition)

Type: Operator (Arithmetic)
Syntax:

- `expr` (unary plus; returns the operand unchanged)
- `expr1  expr2+` (addition; also concatenates strings when both operands are strings, or arrays when both operands are arrays)

Description: Adds two numeric operands, concatenates two strings, or concatenates two arrays element-wise in order. Unary + is allowed for clarity on positive numeric literals.
Example:

```
LET sum% = 5 + 3
LET greeting$ = "Hello, " + name$
LET merged%[] = a%[] + b%[]
```

- (Subtraction)

Type: Operator (Arithmetic)
Syntax:

- `-expr` (unary negation)
- `expr1 - expr2` (subtraction)

Description: Negates a numeric operand, or subtracts the right operand from the left.
Example:

```
LET diff% = 10 - 4
LET opposite% = -x%
```

/ (Division)

Type: Operator (Arithmetic)
Syntax: `expr1 / expr2`
Description: Divides the left operand by the right. Integer division promotes to real when needed; dividing by zero raises an error.
Example:

```
LET quotient# = 15 / 4
```

< (Less Than)

Type: Operator (Comparison)
Syntax: `expr1 < expr2`
Description: Returns `TRUE%` when the left operand is less than the right.
Example:

```
IF temperature# < 32 THEN
    PRINT "Freezing"
END IF
```

<= (Less Than or Equal)

Type: Operator (Comparison)
Syntax: `expr1 <= expr2`
Description: Returns `TRUE%` when the left operand is less than or equal to the right.
Example:

```
IF score% <= 59 THEN
    PRINT "Needs improvement"
END IF
```

<> (Not Equal)

Type: Operator (Comparison)
Syntax: `expr1 <> expr2`
Description: Returns `TRUE%` when the operands are not equal.
Example:

```
IF name$ <> "" THEN
    PRINT "Hello, ", name$
END IF
```

= (Equal)

Type: Operator (Comparison)
Syntax: `expr1 = expr2`
Description: Returns `TRUE%` when both operands are equal, `FALSE%` otherwise. Works on numeric scalars, strings, and compatible composite values.
Example:

```
IF score% = 100 THEN
    PRINT "Perfect"
END IF
```

> (Greater Than)

Type: Operator (Comparison)
Syntax: `expr1 > expr2`
Description: Returns `TRUE%` when the left operand is greater than the right.
Example:

```
IF score% > 90 THEN
    PRINT "A range"
END IF
```

>= (Greater Than or Equal)

Type: Operator (Comparison)
Syntax: `expr1 >= expr2`
Description: Returns `TRUE%` when the left operand is greater than or equal to the right.
Example:

```
IF age% >= 18 THEN
    PRINT "Adult"
END IF
```

^ and (Exponentiation)

Type: Operator (Arithmetic)
Syntax:

- `expr1 ^ expr2`
- `expr1  expr2`

Description: Raises the left operand to the power of the right. ^ and are equivalent spellings. Evaluation is right-associative.
Example:

```
LET power# = 2 ^ 8
LET same# = 2 ** 10
```

ABS

Type: Operator (Math)
Syntax: `ABS expr`
Description: Returns the absolute value of a real operand, or the magnitude (modulus) of a complex operand.
Example:

```
PRINT ABS -5        ' 5
PRINT ABS (3+4i)    ' 5
```

ACOS

Type: Function (Trigonometric)
Syntax: `ACOS x#`
Description: Returns the arccosine of `x` in radians. Input must be in range [-1, 1].
Example:

```
LET angle# = ACOS 0.5
PRINT angle#    ' Prints: 1.047... (π/3 radians)
```

ACOSH

Type: Function (Hyperbolic)
Syntax: `ACOSH x#`
Description: Returns the inverse hyperbolic cosine of `x`. Input must be ≥ 1.
Example:

```
LET result# = ACOSH 2.0
```

AND

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 AND expression2`
Description: Bitwise AND operation. Output bit is 1 when both input bits are 1.
Example:

```
LET result% = 12 AND 10
IF (x% > 0) AND (y% < 10) THEN
    PRINT "Valid"
END IF
```

ASC

Type: Operator (String)
Syntax: `ASC string$`
Description: Returns the Unicode scalar value of the first character in `string$` (the value `CHR` accepts for that character). Empty string returns `0`.
Example:

```
LET code% = ASC "A"         ' 65
LET code% = ASC "a"         ' 97
LET emoji% = ASC "😀"
```

ASIN

Type: Function (Trigonometric)
Syntax: `ASIN x#`
Description: Returns the arcsine of `x` in radians. Input must be in range [-1, 1].
Example:

```
LET angle# = ASIN 0.5
PRINT angle#    ' Prints: 0.524... (π/6 radians)
```

ASINH

Type: Function (Hyperbolic)
Syntax: `ASINH x#`
Description: Returns the inverse hyperbolic sine of `x`.
Example:

```
LET result# = ASINH 1.0
```

ATAN

Type: Function (Trigonometric)
Syntax: `ATAN x#`
Description: Returns the arctangent of `x` in radians.
Example:

```
LET angle# = ATAN 1.0
PRINT angle#    ' Prints: 0.785... (π/4 radians)
```

ATANH

Type: Function (Hyperbolic)
Syntax: `ATANH x#`
Description: Returns the inverse hyperbolic tangent of `x`. Input must be in range (-1, 1).
Example:

```
LET result# = ATANH 0.5
```

BIN

Type: Operator (String)
Syntax: `BIN integer%`
Description: Converts an integer to its binary string representation. Negative numbers are represented using two's complement notation (32 bits for 32-bit integers).

Also accepted: `BIN$` (string-sigil spelling).
Example:

```
LET binStr$ = BIN 10     ' "1010"
LET binStr$ = BIN 255    ' "11111111"
LET binStr$ = BIN -1
```

CARG

Type: Function (Complex)
Syntax: `CARG z&`
Description: Returns the argument (phase angle) of a numeric operand in radians. Integer and Real values promote to Complex with zero imaginary part.
Example:

```
LET z& = 1+1i
LET angle# = CARG z&
PRINT angle#    ' Prints: 0.785... (π/4 radians)
```

CBRT

Type: Function (Math)
Syntax: `CBRT x#`
Description: Returns the cube root of `x`.
Example:

```
LET result# = CBRT 27
PRINT result#    ' Prints: 3.0
```

CEIL

Type: Operator (Rounding)
Syntax: `CEIL x#`
Description: Rounds `x` toward positive infinity (∞). Returns the smallest integer ≥ `x#`.
Example:

```
PRINT CEIL 3.2     ' Prints: 4
PRINT CEIL -3.7    ' Prints: -3
```

CHR

Type: Operator (String)
Syntax: `CHR code%`
Description: Returns a one-character string for the Unicode scalar value `code%` (0 through `1114111`). Isolated surrogate values (`55296`–`57343`) are errors.
Example:

```
LET char$ = CHR 65         ' "A"
LET newline$ = CHR 10      ' Newline
LET d$ = CHR 9585
LET grin$ = CHR 128512     ' Grinning face emoji (U+1F600)
```

CONJ

Type: Function (Complex)
Syntax: `CONJ expr`
Description: Returns the complex conjugate (negates the imaginary part). Integer and Real operands promote to Complex.
Example:

```
LET z& = 3+4i
LET conjugate& = CONJ z&
PRINT conjugate&    ' Prints: 3-4i
```

CONTAINS

Type: Operator (Array Search)
Syntax: `array[] CONTAINS value`
Description: Synonym for `INCLUDES`. Checks if an array contains a value. Returns `TRUE%` (-1) if found, `FALSE%` (0) if not found.
Example:

```
IF numbers%[] CONTAINS 5 THEN
    PRINT "Found"
END IF
IF names$[] CONTAINS "Bob" THEN
    PRINT "Found"
END IF
```

COS

Type: Function (Trigonometric)
Syntax: `COS x#`
Description: Returns the cosine of `x` (where `x` is in radians).
Example:

```
LET result# = COS 0
PRINT result#    ' Prints: 1.0
```

COSH

Type: Function (Hyperbolic)
Syntax: `COSH x#`
Description: Returns the hyperbolic cosine of `x`.
Example:

```
LET result# = COSH 0
PRINT result#    ' Prints: 1.0
```

CRSRLIN% • CRSRCOL%

Type: Integer nullary operator (Text I/O)
Syntax: `CRSRLIN% · CRSRCOL%`
Description: Returns the current text cursor **row** (`CRSRLIN%`) or **column** (`CRSRCOL%`) as 0-based integers on the `PRINT` / `LOCATE` grid (see CRSRLIN% and CRSRCOL%).
Example:

```
LET r% = CRSRLIN%
LET c% = CRSRCOL%
```

DATE$

Type: Nullary Operator (Date/Time)
Syntax: `DATE$`
Description: Nullary operator that returns the current date as a string in `YYYY-MM-DD` format (ISO 8601 date format). This nullary operator is evaluated fresh at runtime, so each evaluation returns the current date.
Example:

```
LET today$ = DATE$
PRINT "Today is: ", today$

IF DATE$ = "2025-12-25" THEN
    PRINT "Merry Christmas!"
END IF
```

DEG

Type: Operator (Unit Conversion)
Syntax: `expression DEG`
Description: Postfix operator that converts degrees to radians.
Example:

```
LET radians# = 90 DEG
PRINT radians#    ' Prints: 1.5708... (π/2 radians)
```

E#

Type: Nullary Operator (Math Constant)
Syntax: E#
Description: Nullary operator that returns the mathematical constant e (Euler's number) as a real number (approximately 2.718281828459045). This nullary operator always returns the same value.
Example:

```
LET eValue# = E#
PRINT eValue#    ' Prints: 2.718281828459045

LET exponential# = E# ^ 2    ' e^2
LET naturalLog# = LOG E#     ' Should be approximately 1.0
```

ENDSWITH

Type: Operator (String)
Syntax: `string$ ENDSWITH suffix$`
Description: Checks if a string ends with the specified suffix. Returns integer: 0 = false, -1 = true. Case-sensitive.
Example:

```
LET filename$ = "document.txt"
IF filename$ ENDSWITH ".txt" THEN
    PRINT "Ends with '.txt'"
END IF
IF filename$ ENDSWITH "doc" THEN
    PRINT "Ends with 'doc'"    ' FALSE
END IF
```

EOF

Type: Operator (File I/O)
Syntax: `EOF fileHandle%`
Description: Unary operator that checks if the file pointer is at the end of the file. Returns integer: 0 = false, -1 = true.
Example:

```
OPEN "data.txt" FOR TEXT READ AS file%
WHILE NOT EOF file%
    READ line$ FROM file%
    PRINT line$
WEND
CLOSE file%
```

EXISTS

Type: Operator (File I/O)
Syntax: `EXISTS "filename"`
Description: Unary operator that checks if a file or directory exists. Returns integer: 0 = false, -1 = true.
Example:

```
IF EXISTS "data.txt" THEN
    READFILE content$ FROM "data.txt"
ELSE
    PRINT "File not found"
END IF
```

EXP

Type: Function (Math)
Syntax: `EXP x#`
Description: Returns e raised to the power `x` (e^x).
Example:

```
LET result# = EXP 1
PRINT result#    ' Prints: 2.718... (e)
```

EXPAND

Type: Operator (Rounding)
Syntax: `EXPAND x#`
Description: Rounds `x` away from zero.
Example:

```
PRINT EXPAND 3.1     ' Prints: 4
PRINT EXPAND -3.1    ' Prints: -4
```

EXTENTS

Type: Operator (Array)
Syntax: `EXTENTS arrayOperand`
Description: Prefix unary operator that returns a **one-dimensional integer array** of per-axis lengths for an array operand. Axis order matches `DIM` and multi-index subscripts (`matrix#[,]` → `[rows, cols]`). For 1D arrays, returns a single-element array (`[n]`). Use `LEN` for total flat element count on multi-dimensional arrays.
Example:

```
DIM matrix#[5, 10]
LET dims%[] = EXTENTS matrix#[,]    ' [5, 10]
FOR i% = 0 TO dims%[0] - 1
    FOR j% = 0 TO dims%[1] - 1
        ' ...
    NEXT j%
NEXT i%
```

FALSE%

Type: Nullary Operator (Boolean Constant)
Syntax: `FALSE%`
Description: Nullary operator that returns the canonical boolean false value as an integer (0). This nullary operator always returns the same value.
Example:

```
LET flag% = FALSE%
IF NOT condition% THEN
    LET result% = FALSE%
END IF
```

```
LET check% = (x% < 0) OR FALSE%    ' Bitwise OR with FALSE%
```

FLOOR

Type: Operator (Rounding)
Syntax: `FLOOR x#`
Description: Rounds `x` toward negative infinity (-∞). Returns the largest integer ≤ `x`.
Example:

```
PRINT FLOOR 3.7      ' Prints: 3
PRINT FLOOR -3.2     ' Prints: -4
```

HEX

Type: Operator (String)
Syntax: `HEX integer%`
Description: Converts an integer to its hexadecimal string representation (uppercase, no prefix). Negative numbers are represented using two's complement notation (8 hex digits for 32-bit integers).

Also accepted: `HEX$` (string-sigil spelling).
Example:

```
LET hexStr$ = HEX 255    ' "FF"
LET hexStr$ = HEX 10     ' "A"
LET hexStr$ = HEX -1     ' "FFFFFFFF"
LET hexStr$ = HEX 4095    ' "FFF"
```

IMAG

Type: Function (Complex)
Syntax: `IMAG z&`
Description: Returns the imaginary component of a numeric operand. Integer and Real values promote to Complex (`IMAG 5` → `0`).
Example:

```
LET z& = 3+4i
LET imagPart# = IMAG z&
PRINT imagPart#    ' Prints: 4.0
```

IMP

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 IMP expression2`
Description: Bitwise implication. Output bit is 1 when first bit is 0 or second bit is 1.
Example:

```
LET result% = 5 IMP 3    ' Binary implication
```

INCLUDES

Type: Operator (Array Search)
Syntax: `array[] INCLUDES value`
Description: Checks if an array includes a value. Returns `TRUE%` (-1) if found, `FALSE%` (0) if not found. `CONTAINS` is an exact synonym and can be used interchangeably.
Example:

```
IF numbers%[] INCLUDES 5 THEN
    PRINT "Found"
END IF
IF names$[] INCLUDES "Bob" THEN
    PRINT "Found"
END IF
```

INDEXOF

Type: Operator (Array Search)
Syntax: `array[] INDEXOF value`
Description: Finds the **0-based** index of the first occurrence of `value` in the array. Returns **`-1`** if not found.
Example:

```
LET index% = numbers%[] INDEXOF 5
IF index% >= 0 THEN
    PRINT "Found at index: ", index%
END IF

LET index% = names$[] INDEXOF "Bob"
IF index% >= 0 THEN
    PRINT "Found at index: ", index%
END IF
```

INKEY$[]

Type: Nullary Operator (Input)
Syntax: `INKEY$[]`
Description: Nullary operator that returns a **one-dimensional string array**: every key **currently held down**, or an empty array if none. Non-blocking; each evaluation allocates a new snapshot of the runtime depressed-key set (no input queue). Elements are either a single-character string or a special key name (see appendix). Use `INKEY$[] INCLUDES "token"` to test membership; special names are **case-sensitive** in string comparisons.

Special key names (elements):

Key	String value
Escape	`"ESC"`
Enter/Return	`"ENTER"`
Backspace	`"BACKSPACE"`
Tab	`"TAB"`
Space	`" "`
Arrow Up	`"ARROWUP"`
Arrow Down	`"ARROWDOWN"`
Arrow Left	`"ARROWLEFT"`
Arrow Right	`"ARROWRIGHT"`
Home	`"HOME"`
End	`"END"`
Page Up	`"PAGEUP"`
Page Down	`"PAGEDOWN"`
Insert	`"INSERT"`
Delete	`"DELETE"`
F1–F12	`"F1"` through `"F12"`
Shift	`"SHIFT"`
Control	`"CTRL"`
Alt	`"ALT"`
Caps Lock	`"CAPSLOCK"`
Command (⌘), macOS	`"CMD"`

Examples:

```
' Poll keyboard; exit when ESC is held
DO
    IF INKEY$[] INCLUDES "ESC" THEN
        EXIT DO
    END IF
```

```
LOOP

' Arrows (each INKEY$[] evaluates current set)
IF INKEY$[] INCLUDES "ARROWUP" THEN
    LET y% += 1
ELSEIF INKEY$[] INCLUDES "ARROWDOWN" THEN
    LET y% -= 1
ELSEIF INKEY$[] INCLUDES "ARROWLEFT" THEN
    LET x% -= 1
ELSEIF INKEY$[] INCLUDES "ARROWRIGHT" THEN
    LET x% += 1
END IF

' Function keys
IF INKEY$[] INCLUDES "F1" THEN
    CALL ShowHelp
ELSEIF INKEY$[] INCLUDES "F2" THEN
    CALL SaveGame
END IF
```

INSTR

Type: Operator (String Search)
Syntax: `needle$ INSTR haystack$` or `needle$ INSTR haystack$ FROM start%`
Description: Finds the first occurrence of `needle$` inside `haystack$`. `FROM start%` is a **0-based** index in the haystack. Returns the **0-based** index of the match, or **`-1`** if not found.
Example:

```
LET pos% = "world" INSTR "Hello world"     ' 6
LET pos% = "o" INSTR "Hello world" FROM 5   ' 7
```

INT

Type: Operator (Math)
Syntax: `INT x#`
Description: Converts a real number to an integer by truncating the decimal part (rounds toward zero).

Also accepted: `CINT` (classic BASIC spelling).
Example:

```
LET dice% = INT (RND# * 6) + 1    ' Random integer 1-6
PRINT INT 3.9    ' Prints: 3
PRINT INT -3.9   ' Prints: -3
```

JOIN

Type: Operator (Array)
Syntax: `array$[] JOIN separator$`
Description: Joins array elements into a string with a separator.
Example:

```
LET names$[] = ["Alice", "Bob", "Charlie"]
LET joined$ = names$[] JOIN ", "    ' "Alice, Bob, Charlie"
```

LASTINDEXOF

Type: Operator (Array Search)
Syntax: `array[] LASTINDEXOF value`
Description: Finds the **0-based** index of the **last** occurrence of `value` in the array. Returns **`-1`** if not found.
Example:

```
LET index% = numbers%[] LASTINDEXOF 5
IF index% >= 0 THEN
    PRINT "Last found at index: ", index%
END IF

LET index% = names$[] LASTINDEXOF "Bob"
IF index% >= 0 THEN
    PRINT "Last found at index: ", index%
END IF
```

LCASE

Type: Operator (String)
Syntax: `LCASE string$`
Description: Returns a copy of the string converted to lowercase.

Also accepted: `LOWER`, `LOWER$`.
Example:

```
LET lower$ = LCASE "WORLD"    ' "world"
```

LEFT

Type: Operator (String)
Syntax: `string$ LEFT length%`
Description: Returns the leftmost `length%` characters of the string. If `length%` is greater than the string length, returns the entire string.
Example:

```
LET text$ = "Hello, world!"
LET firstWord$ = text$ LEFT 5    ' "Hello"
LET firstChar$ = text$ LEFT 1    ' "H"
```

LEN

Type: Operator (String / Array)
Syntax: `LEN expr`
Description: Prefix unary operator. On a string, returns the character count (Unicode scalar values, not UTF-8 bytes). On a 1D array operand (`name%[]`), returns the element count. On a multi-dimensional whole-array operand (`matrix#[,]`), returns the **total** flat element count (product of all axis lengths). For per-axis lengths, use `EXTENTS`.
Example:

```
LET textLen% = LEN message$
LET size% = LEN numbers%[]
LET flat% = LEN matrix#[,]          ' total cells, not per-axis
```

LOC

Type: Operator (File I/O)
Syntax: `LOC fileHandle%`
Description: Unary operator that returns the current byte position in the file (0-based). Supported on `TEXT` handles only; `LOC` on `DATA` raises an error.
Example:

```
OPEN "log.txt" FOR TEXT READ AS file%
LET startPos% = LOC file%
READ line$ FROM file%
LET endPos% = LOC file%
CLOSE file%
```

LOG

Type: Function (Math)
Syntax: `LOG x#`
Description: Returns the natural logarithm (base e) of `x`.
Example:

```
LET result# = LOG 2.718
PRINT result#    ' Prints: 1.0 (approximately)
```

LOG10

Type: Function (Math)
Syntax: `LOG10 x#`
Description: Returns the common logarithm (base 10) of `x`.
Example:

```
LET result# = LOG10 100
PRINT result#    ' Prints: 2.0
```

LOG2

Type: Function (Math)
Syntax: `LOG2 x#`
Description: Returns the binary logarithm (base 2) of `x`.
Example:

```
LET result# = LOG2 8
PRINT result#    ' Prints: 3.0
```

LTRIM

Type: Operator (String)
Syntax: `LTRIM string$`
Description: Returns a copy of the string with leading spaces removed.

Also accepted: `LTRIM$`.
Example:

```
LET trimmed$ = LTRIM "  text"    ' "text"
```

MID

Type: Operator (String)
Syntax: `string$ MID start% TO end%`
Description: Returns a substring from index `start%` to index `end%` (**0-based**, inclusive). If `start%` is beyond the string length, returns an empty string. If `end%` extends beyond the string, returns characters up to the end of the string.
Example:

```
LET text$ = "Hello, world!"
LET world$ = text$ MID 7 TO 11
LET middle$ = text$ MID 2 TO 5
LET char$ = text$ MID 0 TO 0
```

MOD

Type: Operator (Arithmetic)
Syntax: `expression1 MOD expression2`
Description: Returns the remainder of division. Works with both Integer and Real operands. When both operands are integers, the result is an integer. When either operand is real, the result is real.
Example:

```
LET m% = 17 MOD 5
LET m# = 17.5 MOD 5
LET m# = 17 MOD 5.5
LET m# = 17.5 MOD 5.5
PRINT m%
```

NAND

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 NAND expression2`
Description: Bitwise NAND. Output bit is 1 when at least one input bit is 0.
Example:

```
LET result% = 12 NAND 10
```

NOR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 NOR expression2`
Description: Bitwise NOR. Output bit is 1 when both input bits are 0.
Example:

```
LET result% = 12 NOR 10
```

NOT

Type: Operator (Boolean/Bitwise)
Syntax: `NOT expression`
Description: Bitwise NOT. Output bit is 1 when the input bit is 0.
Example:

```
LET result% = NOT 5
IF NOT gameOver% THEN
    CALL UpdateGame
END IF
```

NOTES

Type: Operator (Audio)
Syntax: `NOTES voice%`
Description: Returns the number of entries remaining in the playback queue for the specified voice (per-voice queue, max 65,536 entries). Returns 0 when the voice has finished playing all notes. Useful for checking if a voice is still playing music.
Example:

```
PLAY 1, "CDEFGAB C"
LET notesLeft% = NOTES 1
IF notesLeft% > 0 THEN
    PRINT "Playing: ", notesLeft%, " notes left"
END IF

' Wait for voice to finish
DO
    LET notesLeft% = NOTES 1
LOOP WHILE notesLeft% > 0
PRINT "Voice 1 finished playing"
```

NOW%

Type: Nullary Operator (Date/Time)
Syntax: `NOW%`
Description: Nullary operator that returns the current Unix timestamp (seconds since January 1, 1970, 00:00:00 UTC) as an integer. Useful for calculating time differences and storing absolute time values. This nullary operator is evaluated fresh at runtime, so each evaluation returns the current timestamp.
Example:

```
LET timestamp% = NOW%
PRINT "Timestamp: ", timestamp%

' Calculate elapsed time
LET startTime% = NOW%
SLEEP 2000    ' Wait 2 seconds
LET endTime% = NOW%
LET elapsed% = endTime% - startTime%
PRINT "Elapsed: ", elapsed%, " seconds"
```

OR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 OR expression2`
Description: Bitwise OR. Output bit is 1 when at least one input bit is 1.
Example:

```
LET result% = 12 OR 10    ' Binary: 1100 OR 1010 = 1110 (14)
IF (x% > 0) OR (y% < 10) THEN
    PRINT "Valid"
END IF
```

PI#

Type: Nullary Operator (Math Constant)
Syntax: `PI#`
Description: Nullary operator that returns the mathematical constant π (pi) as a real number (approximately 3.141592653589793). This nullary operator always returns the same value.
Example:

```
LET piValue# = PI#
PRINT piValue#    ' Prints: 3.141592653589793

LET circumference# = 2 * PI# * radius#
LET area# = PI# * radius# * radius#
LET halfCircle# = PI# * radius#
```

RAD

Type: Operator (Unit Conversion)
Syntax: `expression RAD`
Description: Postfix operator that converts radians to degrees.
Example:

```
LET degrees# = PI# RAD
PRINT degrees#    ' Prints: 180.0
```

REAL

Type: Function (Complex)
Syntax: `REAL z&`
Description: Returns the real component of a numeric operand. Integer and Real values promote to Complex (`REAL 42` → `42`).
Example:

```
LET z& = 3+4i
LET re# = REAL z&
PRINT re#    ' Prints: 3.0
```

REPLACE

Type: Operator (String)
Syntax: `string$ REPLACE oldSubstring$ WITH newSubstring$`
Description: Returns a copy of the string with all occurrences of `oldSubstring$` replaced with `newSubstring$`.
Example:

```
LET new$ = "Hello world" REPLACE "world" WITH "EduBASIC"
```

RIGHT

Type: Operator (String)
Syntax: `string$ RIGHT length%`
Description: Returns the rightmost `length%` characters of the string. If `length%` is greater than the string length, returns the entire string.
Example:

```
LET text$ = "Hello, world!"
LET lastWord$ = text$ RIGHT 6    ' "world!"
LET lastChar$ = text$ RIGHT 1    ' "!"
```

RND#

Type: Nullary Operator (Random)
Syntax: `RND#`
Description: Nullary operator that returns a random real number in the range [0, 1). This nullary operator is evaluated fresh at runtime, so each evaluation returns a different random value.
Example:

```
LET random# = RND#
LET dice% = INT (RND# * 6) + 1    ' Random integer 1-6
```

ROUND

Type: Operator (Rounding)
Syntax: `ROUND x#`
Description: Rounds `x` to the nearest integer. Ties round up.
Example:

```
PRINT ROUND 3.5      ' Prints: 4
PRINT ROUND 3.4      ' Prints: 3
```

RTRIM

Type: Operator (String)
Syntax: `RTRIM string$`
Description: Returns a copy of the string with trailing spaces removed.

Also accepted: `RTRIM$`.
Example:

```
LET trimmed$ = RTRIM "text  "    ' "text"
```

SGN

Type: Function (Math)
Syntax: `SGN x#`
Description: Returns the sign of `x`: -1 (negative), 0 (zero), or 1 (positive).
Example:

```
PRINT SGN 5        ' Prints: 1
PRINT SGN -3       ' Prints: -1
PRINT SGN 0        ' Prints: 0
```

SIN

Type: Function (Trigonometric)
Syntax: `SIN x#`
Description: Returns the sine of `x` (where `x` is in radians).
Example:

```
LET result# = SIN (90 DEG)
PRINT result#    ' Prints: 1.0
```

SINH

Type: Function (Hyperbolic)
Syntax: `SINH x#`
Description: Returns the hyperbolic sine of `x`.
Example:

```
LET result# = SINH 0
PRINT result#    ' Prints: 0.0
```

SQRT

Type: Function (Math)
Syntax: `SQRT x#`
Description: Returns the square root of `x`. For negative real operands, the result is complex (principal branch). Complex operands use the principal complex square root.
Example:

```
LET result# = SQRT 16
PRINT result#    ' Prints: 4.0

LET root& = SQRT (-1+0i)
PRINT root&    ' Prints: 0+1i
```

STARTSWITH

Type: Operator (String)
Syntax: `string$ STARTSWITH prefix$`
Description: Checks if a string starts with the specified prefix. Returns integer: 0 = false, -1 = true. Case-sensitive.
Example:

```
LET filename$ = "document.txt"
IF filename$ STARTSWITH "doc" THEN
    PRINT "Starts with 'doc'"
END IF
IF filename$ STARTSWITH ".txt" THEN
    PRINT "Starts with '.txt'"    ' FALSE%
END IF
```

STR

Type: Operator (Serialization)
Syntax: `STR expression`
Description: Serializes any value to one **EBON** line using the same literal spelling as the Code editor, `CONSOLE`, and the Console REPL. This matches DATA `WRITE` / `READ` lines and pairs with `VAL` for round-trip storage.

Also accepted: `CSTR`, `STR$`.
Example:

```
LET num% = 42
LET numStr$ = STR num%
LET piStr$ = STR 3.14159
LET complexStr$ = STR (3+4i)
LET complexStr$ = STR (3-4i)
```

TAN

Type: Function (Trigonometric)
Syntax: `TAN x#`
Description: Returns the tangent of `x` (where `x` is in radians).
Example:

```
LET result# = TAN (45 DEG)
PRINT result#    ' Prints: 1.0
```

TANH

Type: Function (Hyperbolic)
Syntax: `TANH x#`
Description: Returns the hyperbolic tangent of `x`.
Example:

```
LET result# = TANH 0
PRINT result#    ' Prints: 0.0
```

TIME$

Type: Nullary Operator (Date/Time)
Syntax: `TIME$`
Description: Nullary operator that returns the current time as a string in `HH:MM:SS` format (24-hour format). This nullary operator is evaluated fresh at runtime, so each evaluation returns the current time.
Example:

```
LET now$ = TIME$
PRINT "Time: ", now$

PRINT DATE$, " ", TIME$, " - Program started"
```

TRIM

Type: Operator (String)
Syntax: `TRIM string$`
Description: Returns a copy of the string with leading and trailing spaces removed.

Also accepted: `TRIM$`.
Example:

```
LET trimmed$ = TRIM "  text  "    ' "text"
```

TRUE%

Type: Nullary Operator (Boolean Constant)
Syntax: `TRUE%`
Description: Nullary operator that returns the canonical boolean true value as an integer (-1). All bits are set to 1 in two's complement representation, which makes bitwise operations behave correctly. For example, `TRUE% AND TRUE%` yields `TRUE%` (`-1 AND -1 = -1`). This nullary operator always returns the same value.
Example:

```
LET flag% = TRUE%
IF condition% THEN
    LET result% = TRUE%
END IF

LET check% = (x% > 0) AND TRUE%    ' Bitwise AND with TRUE%
```

TRUNC

Type: Operator (Rounding)
Syntax: `TRUNC x#`
Description: Rounds `x` toward zero (truncates the decimal part).
Example:

```
PRINT TRUNC 3.9      ' Prints: 3
PRINT TRUNC -3.9     ' Prints: -3
```

UCASE

Type: Operator (String)
Syntax: `UCASE string$`
Description: Returns a copy of the string converted to uppercase.

Also accepted: `UPPER`, `UPPER$`.
Example:

```
LET upper$ = UCASE "hello"    ' "HELLO"
```

VAL

Type: Operator (Deserialization)
Syntax: `VAL expression`
Description: Parses a string as **one EBON literal** (same rules as DATA `READ`; multi-line documents are supported). The text must be a single literal only (no variables or calls). Invalid or non-literal text raises an error. Non-string operands pass through unchanged.
Example:

```
LET text$ = "123"
LET number% = VAL text$
LET decimal$ = "3.14"
LET value# = VAL decimal$
LET hexValue% = VAL "&HFF"
LET binValue% = VAL "&B1010"
LET complexValue& = VAL "3+4i"
LET complexValue& = VAL "3-4i"
LET row$ = STR myStruct
LET copyStruct = VAL row$
```

XNOR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 XNOR expression2`
Description: Bitwise XNOR. Output bit is 1 when both input bits are the same.
Example:

```
LET result% = 12 XNOR 10
```

XOR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 XOR expression2`
Description: Bitwise XOR. Output bit is 1 when the input bits are different.
Example:

```
LET result% = 12 XOR 10
```

Appendix: MML Syntax

These tokens are for the string you pass to **PLAY**. Parsing is **left to right**. Letters may be **upper or lower** case. Put **digits directly** after `O`, `L`, `V`, `R`, `N`, and after a note letter when you give a length (for example `C4`, not `C 4`). Any character that does not start a known token is **skipped**. For each **voice**, the scanner keeps the current octave, default length (`L`), velocity (`V`), and articulation (`MN` / `ML` / `MS`); that state is updated as tokens are parsed and as queued `O` / `L` / `V` / `MN` / `ML` / `MS` / **>** / < entries are processed, and it carries over to the next `PLAY` on that voice (each voice is independent).

Token	Role
`A` … `G`	Note. Optional `#`, `+` or `-` right after the letter. Optional **length** digits, then optional **.** (one or more for dotted / double-dotted). If you omit digits, the current default length applies; you can still add dots (e.g. `L4` `C.`).
`N` + digits	MIDI note number (0–127). Optional length digits and `.` (same dotted rules).
`R`	Rest. Optional length digits and `.` (uses the current default length if you omit digits).
`L` + digits	Default note/rest length (for example `L4` = quarter, `L8` = eighth).
`O` + digits	Octave **0–9** (starts at **4**).
> / <	Raise or lower octave by **1** (stays within 0–9).
`V` + digits	Velocity **0–127** for following notes (starts at **64**).
`MN`	Music normal articulation (default): each note sounds **7/8** of its written slot.
`ML`	Music legato: each note sounds the **full** written slot.
`MS`	Music staccato: each note sounds **3/4** of its written slot.

Duration: For length denominator **n** (the number after `L` or after a note / `R` / `N`), the base duration is **4/n** beats (a quarter note when **n** = 4). With **d** dot characters after the optional digits, duration is **(4/n) × (2 – 2^–d^)** beats (so **d** = 0 gives the base; **d** = 1 gives **6/n**; **d** = 2 gives **7/n**). One **beat** lasts **60 / (TEMPO in BPM)** seconds at the time the note or rest plays. Articulation (`MN` / `ML` / `MS`) scales only the **sounding** portion within that slot; the slot length itself is unchanged.

Queue size: `O`, `L`, `V`, `MN`, `ML`, `MS`, **>**, <, each note, and each rest each add **one** queue entry (maximum **65,536** per voice).

Appendix: General MIDI Instruments

`VOICE ... INSTRUMENT` accepts a program **number** (**1–128**) or a **name** in quotes. Names are **not** case-sensitive (**exact** match, then **substring** match; otherwise **1**, Acoustic Grand Piano). Use **Name** from the table with `INSTRUMENT "..."`.

Prog	Name
1	Acoustic Grand Piano
2	Bright Acoustic Piano
3	Electric Grand Piano
4	Honky-tonk Piano
5	Electric Piano 1
6	Electric Piano 2
7	Harpsichord
8	Clavi
9	Celesta
10	Glockenspiel
11	Music Box
12	Vibraphone
13	Marimba
14	Xylophone
15	Tubular Bells
16	Dulcimer
17	Drawbar Organ
18	Percussive Organ
19	Rock Organ
20	Church Organ
21	Reed Organ
22	Accordion
23	Harmonica
24	Tango Accordion
25	Acoustic Guitar (nylon)
26	Acoustic Guitar (steel)
27	Electric Guitar (jazz)
28	Electric Guitar (clean)
29	Electric Guitar (muted)
30	Overdriven Guitar

Prog	Name
31	Distortion Guitar
32	Guitar Harmonics
33	Acoustic Bass
34	Electric Bass (finger)
35	Electric Bass (pick)
36	Fretless Bass
37	Slap Bass 1
38	Slap Bass 2
39	Synth Bass 1
40	Synth Bass 2
41	Violin
42	Viola
43	Cello
44	Contrabass
45	Tremolo Strings
46	Pizzicato Strings
47	Orchestral Harp
48	Timpani
49	String Ensemble 1
50	String Ensemble 2
51	Synth Strings 1
52	Synth Strings 2
53	Choir Aahs
54	Voice Oohs
55	Synth Voice
56	Orchestra Hit
57	Trumpet
58	Trombone
59	Tuba
60	Muted Trumpet
61	French Horn
62	Brass Section
63	Synth Brass 1
64	Synth Brass 2
65	Soprano Sax
66	Alto Sax
67	Tenor Sax
68	Baritone Sax

Prog	Name
69	Oboe
70	English Horn
71	Bassoon
72	Clarinet
73	Piccolo
74	Flute
75	Recorder
76	Pan Flute
77	Blown Bottle
78	Shakuhachi
79	Whistle
80	Ocarina
81	Lead 1 (square)
82	Lead 2 (sawtooth)
83	Lead 3 (calliope)
84	Lead 4 (chiff)
85	Lead 5 (charang)
86	Lead 6 (voice)
87	Lead 7 (fifths)
88	Lead 8 (bass + lead)
89	Pad 1 (new age)
90	Pad 2 (warm)
91	Pad 3 (polysynth)
92	Pad 4 (choir)
93	Pad 5 (bowed)
94	Pad 6 (metallic)
95	Pad 7 (halo)
96	Pad 8 (sweep)
97	FX 1 (rain)
98	FX 2 (soundtrack)
99	FX 3 (crystal)
100	FX 4 (atmosphere)
101	FX 5 (brightness)
102	FX 6 (goblins)
103	FX 7 (echoes)
104	FX 8 (sci-fi)
105	Sitar
106	Banjo

Prog	Name
107	Shamisen
108	Koto
109	Kalimba
110	Bag pipe
111	Fiddle
112	Shanai
113	Tinkle Bell
114	Agogo
115	Steel Drums
116	Woodblock
117	Taiko Drum
118	Melodic Tom
119	Synth Drum
120	Reverse Cymbal
121	Guitar Fret Noise
122	Breath Noise
123	Seashore
124	Bird Tweet
125	Telephone Ring
126	Helicopter
127	Applause
128	Gunshot

More detail: General MIDI Level 1 (MIDI Association). Spelling can vary slightly by synthesizer; use **exact** names from this table when you care which patch you get.

The shipped **GeneralUser GS** bank also contains **many additional melodic presets** (layers, synth waves, GS variants) on auxiliary MIDI banks. See Appendix: Drum Kits and Appendix: Auxiliary Melodic Banks (GM program numbers stay in Appendix: General MIDI Instruments).

Appendix: General MIDI Standard Percussion Map

When a voice uses a **drum kit** preset, **MIDI note numbers** select **percussion sounds** (General MIDI Level 1 **standard kit**). In MML, spell these as **`N`** followed by the number (see Percussion and Drum Kits). This table is the usual **note → sound** map; exact samples come from the SoundFont.

Note names use the same convention as MML: **middle C = C4 = MIDI note 60**. Drum parts still use **`N`** with the number; the **Name** column is for players who read keyboard pitch (e.g. in a DAW drum map).

Note	Name	Percussion sound
35	B1	Acoustic Bass Drum
36	C2	Bass Drum 1
37	C#2	Side Stick
38	D2	Acoustic Snare
39	D#2	Hand Clap
40	E2	Electric Snare
41	F2	Low Floor Tom
42	F#2	Closed Hi-Hat
43	G2	High Floor Tom
44	G#2	Pedal Hi-Hat
45	A2	Low Tom
46	A#2	Open Hi-Hat
47	B2	Low-Mid Tom
48	C3	Hi-Mid Tom
49	C#3	Crash Cymbal 1
50	D3	High Tom
51	D#3	Ride Cymbal 1
52	E3	Chinese Cymbal
53	F3	Ride Bell
54	F#3	Tambourine
55	G3	Splash Cymbal
56	G#3	Cowbell
57	A3	Crash Cymbal 2
58	A#3	Vibraslap
59	B3	Ride Cymbal 2
60	C4	Hi Bongo
61	C#4	Low Bongo

Note	Name	Percussion sound
62	D4	Mute Hi Conga
63	D#4	Open Hi Conga
64	E4	Low Conga
65	F4	High Timbale
66	F#4	Low Timbale
67	G4	High Agogo
68	G#4	Low Agogo
69	A4	Cabasa
70	A#4	Maracas
71	B4	Short Whistle
72	C5	Long Whistle
73	C#5	Short Guiro
74	D5	Long Guiro
75	D#5	Claves
76	E5	Hi Wood Block
77	F5	Low Wood Block
78	F#5	Mute Cuica
79	G5	Open Cuica
80	G#5	Mute Triangle
81	A5	Open Triangle

Authoritative overview: General MIDI Level 1 (MIDI Association).

Appendix: Drum Kits

The app ships **GeneralUser GS**. `VOICE … INSTRUMENT "…"` tries **drum-kit** preset names first across the loaded bank; for the GM **`N`** percussion map once a kit is loaded, see Percussion and Drum Kits. It then resolves **melodic** names: **GM bank 0** (programs **1–128**) first, then **any other non-drum** preset in the file (see Appendix: Auxiliary Melodic Banks). Matching is **case-insensitive** and allows **substring** matches, so short queries (e.g. `"jazz"`) may pick the first drum preset whose name contains that text, use a **longer, exact** string when you care which kit you get.

The **drum** preset names in the table below (and close variants) appear in the shipped sound bank and work the same way as **`"Standard 1 Kit"`** for GM-style **note → drum** mapping. Spelling must match closely enough for EduBASIC's name lookup; if one form fails, try the other form in the same row.

Preset	Remarks
`"Standard 1 Kit"`	Default-style GM kit
`"Standard 2 Kit"` / `"Standard 2"`	Alternate standard kit
`"Standard 1"` / `"Standard 3"`	Shorter bank names for standard kits
`"Jazz Kit"`	Jazz-oriented mapping
`"Brush Kit"`	Brush/snare emphasis
`"Orchestral Kit"` / `"Orchestral"`	Orchestral percussion set
`"SFX Kit"`	Sound-effects oriented kit
`"CM-64/32L Kit"` / `"CM-64/32L"`	CM-style kit
`"Room"`	Room-style kit
`"Power"`	Heavier rock-style kit
`"Electronic"`	Electronic kit
`"808/909"`	Classic drum-machine style
`"Accessory Drums 1"` / `"Accessory Drums 2"`	Extra accessory percussion sets

GeneralUser GS ships **thirteen** drum kits; spelling can differ slightly between SoundFont revisions. If a name in this guide does not match, see the GeneralUser GS preset list.

Appendix: Auxiliary Melodic Banks

For **melodic** presets, Appendix: General MIDI Instruments already documents GM **program** numbers **1–128** (Level 1 channel timbres). The table below lists **only** the **extra** GeneralUser GS presets on **non-zero** banks (same program change as that appendix, different bank). For the **bank 0** spelling GeneralUser uses for each program (e.g. **Grand Piano** for program **1**), see section 4.1 of the author's preset list, many names align closely with the GM appendix.

GeneralUser GS adds **melodic** presets on **banks other than 0** while reusing the same **MIDI program** numbers (**1–128**) as that appendix. **`INSTRUMENT` numbers** always select **bank 0** only. **`INSTRUMENT "…"`** (melodic) tries **bank 0** names first, then matches these auxiliary presets (and any other non-drum name in the loaded SoundFont).

```
VOICE 1 INSTRUMENT "Mandolin"
VOICE 2 INSTRUMENT "Tin Whistle"
VOICE 3 INSTRUMENT "Full Orchestra"
TEMPO 90
PLAY 1, "L4 G B D G5"
PLAY 2, "L4 D F# A D5"
PLAY 3, "L2 C G C5"
```

PC is the GM program number (same numbering as the GM instrument appendix). **Bank** is MIDI bank-select MSB in the GeneralUser GS layout. The **Quoted name** column matches the string you pass to **`INSTRUMENT "…"`** (same quoting style as Appendix: Drum Kits).

PC	Bank	Preset
0	11	`"Piano & Str.-Fade"`
0	12	`"Bell Piano"`
1	11	`"Piano & Str.-Sus"`
4	8	`"Chorused Tine EP"`
4	11	`"Tine & FM EPs"`
4	12	`"Bell Tine EP"`
5	8	`"Chorused FM EP"`
5	11	`"Piano & FM EP"`
6	8	`"Coupled Harpsichord"`
6	11	`"Harpsichord noVel"`
6	12	`"Coupled Harpsi noVel"`
8	11	`"Tinkling Bells"`
10	12	`"Christmas Bells"`
11	11	`"Vibraphone No Trem."`

PC	Bank	Preset
14	8	"Church Bells"
14	9	"Carillon"
14	11	"Bell Tower"
16	8	"Detuned Tnwl. Organ"
16	11	"Tonewheel Org noVel"
16	12	"Detun Tnwl Org noVel"
17	8	"Detuned Perc. Organ"
17	11	"Percussive Org noVel"
17	12	"Detun Perc Org noVel"
18	11	"Rock Organ noVel"
19	8	"Pipe Organ 2"
19	11	"Pipe Organ noVel"
19	12	"Pipe Organ 2 noVel"
20	11	"Reed Organ noVel"
21	8	"Italian Accordion"
24	8	"Ukulele"
25	8	"12-String Guitar"
25	16	"Mandolin"
26	8	"Hawaiian Guitar"
27	8	"Chorused Clean Gt."
27	12	"Clean Guitar 2"
28	8	"Funk Guitar"
29	11	"Wah Guitar (CC21)"
30	8	"Feedback Guitar"
31	8	"Guitar Feedback"
38	1	"Synth Bass 101"
38	8	"Acid Bass"
38	11	"Techno Bass"
38	12	"Mean Saw Bass"
39	8	"Beef FM Bass"
39	11	"Pulse Bass"
48	1	"Fast Strings Mono"
48	8	"Orchestra Pad"
48	12	"Full Orchestra"
48	13	"Woodwind Choir"
49	1	"Slow Strings Mono"
49	11	"Velo Strings"
49	12	"Velo Strings Mono"

PC	Bank	Preset
50	8	"Synth Strings 3"
50	11	"Synth Strings 4"
51	11	"Synth Strings 5"
52	1	"Concert Choir Mono"
56	1	"Trumpet 2"
57	1	"Trombone 2"
60	1	"Solo French Horn"
61	1	"Brass Section Mono"
61	8	"Brass Section 2"
61	11	"Brass Section 3"
62	8	"Synth Brass 3"
63	8	"Synth Brass 4"
75	24	"Tin Whistle"
75	25	"Tin Whistle Nm"
75	26	"Tin Whistle Or"
78	11	"Whistlin'"
80	1	"Square Wave"
80	8	"Sine Wave"
80	12	"Square Lead 2"
80	13	"Square Lead 3"
81	1	"Saw Wave"
81	8	"Doctor Solo"
81	11	"Sawtooth Stab"
81	12	"Saw Lead 2"
81	13	"Saw Lead 3"
88	11	"Harpsi Pad"
88	12	"Fantasia 2"
88	13	"Night Vision"
89	11	"Solar Wind"
89	12	"Solar Wind 2"
96	11	"Mystery Pad"
98	1	"Synth Mallet"
98	11	"Synth Chime"
100	11	"Bright Saw Stack"
102	2	"Echo Pan"
107	8	"Taisho Koto"
115	8	"Castanets"
116	8	"Concert Bass Drum"

PC	Bank	Preset
117	8	"Melodic Tom 2"
118	8	"808 Tom"
119	11	"Cymbal Crash"
119	12	"Tambourine"
120	1	"Cut Noise"
120	2	"String Slap"
121	1	"Fl. Key Click"
121	11	"Filter Snap"
122	1	"Rain"
122	2	"Thunder"
122	3	"Wind"
122	4	"Stream"
122	5	"Bubbles"
122	11	"Howling Winds"
122	12	"White Noise Wave"
123	1	"Dog"
123	2	"Horse Gallop"
123	3	"Bird 2"
124	1	"Telephone 2"
124	2	"Door Creaking"
124	3	"Door"
124	4	"Scratch"
124	5	"Windchime"
125	1	"Car-Engine"
125	2	"Car-Stop"
125	3	"Car-Pass"
125	4	"Car-Crash"
125	5	"Siren"
125	6	"Train"
125	7	"Jet Plane"
125	8	"Starship"
125	9	"Burst Noise"
126	1	"Laughing"
126	2	"Scream"
126	3	"Punch"
126	4	"Heart Beat"
126	5	"Footsteps"
127	1	"Machine Gun"

PC	Bank	Preset
127	2	"Lasergun"
127	3	"Explosion"
127	11	"Interference"
127	12	"Shooting Star"

Appendix: INKEY$[] Special Key

When a non-printing key is held, `INKEY$[]` may contain one of these **exact** strings as an element (otherwise you get a normal character such as `"a"` or `" "`):

Key	String in `INKEY$[]`
Escape	`"ESC"`
Enter / Return	`"ENTER"`
Backspace	`"BACKSPACE"`
Tab	`"TAB"`
Space	`" "`
Arrow Up	`"ARROWUP"`
Arrow Down	`"ARROWDOWN"`
Arrow Left	`"ARROWLEFT"`
Arrow Right	`"ARROWRIGHT"`
Home	`"HOME"`
End	`"END"`
Page Up	`"PAGEUP"`
Page Down	`"PAGEDOWN"`
Insert	`"INSERT"`
Delete	`"DELETE"`
F1–F12	`"F1"` … `"F12"`
Shift	`"SHIFT"`
Control	`"CTRL"`
Alt	`"ALT"`
Caps Lock	`"CAPSLOCK"`
Command (⌘), macOS	`"CMD"`

Appendix: CSS Color Names

EduBASIC supports all standard CSS color names as preset colors. Color names are case-insensitive and can be used anywhere a **solid** 32-bit RGBA color value is accepted, including the `COLOR` statement and graphics statements whose **`WITH`** operand is a solid pen or fill color (`PSET`, `LINE`, `ARC`, outline shapes, and **`FILLED`** interiors when not using a sprite array).

Usage:

- `COLOR "red"`: Use color name directly (string expressions are automatically recognized as color names)
- `PSET (100, 100) WITH "blue"`: Use color name in graphics statements
- `PAINT (50, 50) WITH "gold"`: Named color in flood fill
- Color names resolve to their standard CSS color values in `&HRRGGBBAA` format with full opacity (alpha = 255)
- **Sprite array `WITH`** fills use packed integers only; **CSS names do not apply** to the array itself (they still apply when **`WITH`** is a string color)

Aliases:

- `gray` and `grey` are both supported (same color)
- `aqua` and `cyan` are both supported (same color)
- `fuchsia` and `magenta` are both supported (same color)
- `darkgray/darkgrey`, `dimgray/dimgrey`, `lightgray/lightgrey`, `lightslategray/lightslategrey`, `slategray/slategrey`, and `darkslategray/darkslategrey` are all supported

The following table lists all 147 standard CSS color names in alphabetical order:

Color Name	Value (&HRRGGBBAA)	Swatch
aliceblue	&HF0F8FFFF	
antiquewhite	&HFAEBD7FF	
aqua	&H00FFFFFF	
aquamarine	&H7FFFD4FF	
azure	&HF0FFFFFF	

Color Name	Value (&HRRGGBBAA)	Swatch
beige	&HF5F5DCFF	
bisque	&HFFE4C4FF	
black	&H000000FF	
blanchedalmond	&HFFEBCDFF	
blue	&H0000FFFF	
blueviolet	&H8A2BE2FF	
brown	&HA52A2AFF	
burlywood	&HDEB887FF	
cadetblue	&H5F9EA0FF	
chartreuse	&H7FFF00FF	
chocolate	&HD2691EFF	
coral	&HFF7F50FF	
cornflowerblue	&H6495EDFF	
cornsilk	&HFFF8DCFF	
crimson	&HDC143CFF	
cyan	&H00FFFFFF	
darkblue	&H00008BFF	
darkcyan	&H008B8BFF	
darkgoldenrod	&HB8860BFF	
darkgray	&HA9A9A9FF	
darkgreen	&H006400FF	
darkkhaki	&HBDB76BFF	
darkmagenta	&H8B008BFF	
darkolivegreen	&H556B2FFF	
darkorange	&HFF8C00FF	
darkorchid	&H9932CCFF	

Color Name	Value (&HRRGGBBAA)	Swatch
darkred	&H8B0000FF	
darksalmon	&HE9967AFF	
darkseagreen	&H8FBC8FFF	
darkslateblue	&H483D8BFF	
darkslategray	&H2F4F4FFF	
darkturquoise	&H00CED1FF	
darkviolet	&H9400D3FF	
deeppink	&HFF1493FF	
deepskyblue	&H00BFFFFF	
dimgray	&H696969FF	
dodgerblue	&H1E90FFFF	
firebrick	&HB22222FF	
floralwhite	&HFFFAF0FF	
forestgreen	&H228B22FF	
fuchsia	&HFF00FFFF	
gainsboro	&HDCDCDCFF	
ghostwhite	&HF8F8FFFF	
gold	&HFFD700FF	
goldenrod	&HDAA520FF	
gray	&H808080FF	
green	&H008000FF	
greenyellow	&HADFF2FFF	
honeydew	&HF0FFF0FF	
hotpink	&HFF69B4FF	
indianred	&HCD5C5CFF	
indigo	&H4B0082FF	

Color Name	Value (&HRRGGBBAA)	Swatch
ivory	&HFFFFF0FF	
khaki	&HF0E68CFF	
lavender	&HE6E6FAFF	
lavenderblush	&HFFF0F5FF	
lawngreen	&H7CFC00FF	
lemonchiffon	&HFFFACDFF	
lightblue	&HADD8E6FF	
lightcoral	&HF08080FF	
lightcyan	&HE0FFFFFF	
lightgoldenrodyellow	&HFAFAD2FF	
lightgray	&HD3D3D3FF	
lightgreen	&H90EE90FF	
lightpink	&HFFB6C1FF	
lightsalmon	&HFFA07AFF	
lightseagreen	&H20B2AAFF	
lightskyblue	&H87CEFAFF	
lightslategray	&H778899FF	
lightsteelblue	&HB0C4DEFF	
lightyellow	&HFFFFE0FF	
lime	&H00FF00FF	
limegreen	&H32CD32FF	
linen	&HFAF0E6FF	
magenta	&HFF00FFFF	
maroon	&H800000FF	
mediumaquamarine	&H66CDAAFF	
mediumblue	&H0000CDFF	

Color Name	Value (&HRRGGBBAA)	Swatch
mediumorchid	&HBA55D3FF	
mediumpurple	&H9370DBFF	
mediumseagreen	&H3CB371FF	
mediumslateblue	&H7B68EEFF	
mediumspringgreen	&H00FA9AFF	
mediumturquoise	&H48D1CCFF	
mediumvioletred	&HC71585FF	
midnightblue	&H191970FF	
mintcream	&HF5FFFAFF	
mistyrose	&HFFE4E1FF	
moccasin	&HFFE4B5FF	
navajowhite	&HFFDEADFF	
navy	&H000080FF	
oldlace	&HFDF5E6FF	
olive	&H808000FF	
olivedrab	&H6B8E23FF	
orange	&HFFA500FF	
orangered	&HFF4500FF	
orchid	&HDA70D6FF	
palegoldenrod	&HEEE8AAFF	
palegreen	&H98FB98FF	
paleturquoise	&HAFEEEEFF	
palevioletred	&HDB7093FF	
papayawhip	&HFFEFD5FF	
peachpuff	&HFFDAB9FF	
peru	&HCD853FFF	

Color Name	Value (&HRRGGBBAA)	Swatch
pink	&HFFC0CBFF	
plum	&HDDA0DDFF	
powderblue	&HB0E0E6FF	
purple	&H800080FF	
rebeccapurple	&H663399FF	
red	&HFF0000FF	
rosybrown	&HBC8F8FFF	
royalblue	&H4169E1FF	
saddlebrown	&H8B4513FF	
salmon	&HFA8072FF	
sandybrown	&HF4A460FF	
seagreen	&H2E8B57FF	
seashell	&HFFF5EEFF	
sienna	&HA0522DFF	
silver	&HC0C0C0FF	
skyblue	&H87CEEBFF	
slateblue	&H6A5ACDFF	
slategray	&H708090FF	
snow	&HFFFAFAFF	
springgreen	&H00FF7FFF	
steelblue	&H4682B4FF	
tan	&HD2B48CFF	
teal	&H008080FF	
thistle	&HD8BFD8FF	
tomato	&HFF6347FF	
turquoise	&H40E0D0FF	

Color Name	Value (&HRRGGBBAA)	Swatch
violet	&HEE82EEFF	
wheat	&HF5DEB3FF	
white	&HFFFFFFFF	
whitesmoke	&HF5F5F5FF	
yellow	&HFFFF00FF	
yellowgreen	&H9ACD32FF	

Note: The swatches in the table above are visual representations. In actual usage, color names are case-insensitive, so `"Red"`, `"red"`, and `"RED"` all refer to the same color.

Appendix: Turtle Commands

Tokens are **case-insensitive** and separated by **whitespace**. **FD**, **BK**, **RT**, and **LT** must be followed by a **number** (the next token).

Command	Action
`FD n`	Move forward `n` pixels (draw if the pen is down).
`BK n`	Move backward `n` pixels.
`RT n`	Turn right `n` degrees.
`LT n`	Turn left `n` degrees.
`PU`	Pen up (moving does not draw).
`PD`	Pen down (moving draws).
`HOME`	Go to the canvas center; heading becomes **90°** (up).