QUADBASIC LANGUAJE REFERENCE

Table of Contents

Type System and Storage

QuadBasic is a microscope over the 4004: every type maps to concrete registers or RAM nibbles listed in the **Variable Map** at the top of generated ASM.

Philosophy

QuadBasic is not an abstraction layer that hides the Intel 4004. **QuadBasic is a microscope over the 4004.** Every type, every operator, every statement must translate to assembly code that is recognisable as something a 1971 programmer could have written by hand.

This single principle drives the design of the type system:

- `A + B` in source **maps to** the corresponding `ADD` instruction in the generated ASM.
- `A * B` and `A / B` in source are **valid QuadBasic expressions**; the generated ASM uses repeated `ADD` and `SUB` loops, not hidden multiply or divide instructions.
- A `BYTE` addition **maps to** manual carry propagation between two nibbles.
- A `BCD` addition **maps to** the `DAA` instruction, which is the very reason Intel designed the 4004 the way they did.
- A declared `BYTE` variable **appears** in the Variable Map with the exact RAM 4002 nibbles that hold its high and low digits.

When a feature would either hide a 4004 mechanism (no manual carry, no visible memory map, no explicit type for decimal arithmetic) or invent a mechanism the 4004 does not have (signed integers, floating point, pointer arithmetic), that feature is rejected. **A QuadBasic program is a guided tour through the 4004 instruction set, not a way to forget it exists.**

The cost of this principle is verbosity. A `BYTE` addition is half a page of ASM. That is by design: the verbosity is the point.

The Type System at a Glance

QuadBasic defines four explicit scalar/array families. Every variable in a program is declared with a `DIM` statement and a type. There are no implicit types and no untyped variables.

Type	Bits	Range	Default Storage	Role
`NIBBLE`	4	0..15	CPU register (R0..R9)	Native 4004 arithmetic; `ADD Rn`, `XCH`, bitwise ops
`BYTE`	8	0..255	RAM 4002 (2 nibbles)	Manual carry propagation between nibbles
`BCD(n)`	4n	0.. (10^n-1)	RAM 4002 (n nibbles)	The `DAA` instruction; the original Busicom mission of the 4004
`<T> ARRAY(N)`	N × size(T)	–	RAM 4002 (contiguous)	Dynamic addressing with `FIM`/`SRC`

Every type has the following well-defined properties:

- A **size in nibbles**, fixed at compile time.
- A **default storage location** chosen automatically by the compiler.
- A set of **legal operators** (arithmetic, logic, comparison).
- A set of **legal conversions** to and from other types.

There is no notion of an "untyped" or "polymorphic" variable. Every name in a QuadBasic program has exactly one type, fixed for the lifetime of the program.

Choosing a numeric type

QuadBasic defines three scalar numeric families. **Pick the type by what you are doing**, not by how large the number looks in decimal.

Your goal	Use	Why
Fast counter 0..15, LED digit, native register	**NIBBLE**	Lives in R0..R9; `FOR` 0..15
Game logic, VDP coordinates, loop 0..255, small binary math	**BYTE**	8-bit program integer in RAM; `FOR` 0..255; `*`, `/`
Values shown or entered as decimal (calculator, score)	**BCD(n)**	One decimal digit per nibble; `ADD` + `DAA`
ROM address or 12-bit pointer	**NIBBLE** pair + `PEEK @ROM` / `BYTE` + **BCD(n)** for large literals	The 4004 exposes addresses as HI/LO nibbles; QuadBasic has no separate 12-bit scalar type

Rules that keep programs predictable:

- Do **not** mix BCD with NIBBLE/BYTE without an explicit cast (`BCD4(x)`, `BYTE(x)`, …).
- `FOR` / `NEXT` accept **NIBBLE** (0..15, STEP -8..7) or **BYTE** (0..255, STEP -255..255). They are not available on `BCD`.

Arithmetic in Source and in ASM

QuadBasic and the Intel 4004 answer different questions. **QuadBasic defines what you may write in source. The 4004 defines which instructions appear in the generated ASM.**

In QuadBasic source

Type	`+` `-`	`*` `/`	`FOR` / `NEXT`
NIBBLE	Yes	Yes (register-sized)	Yes, 0..15
BYTE	Yes	Yes	Yes, 0..255
BCD(n)	Yes	**No**	**No**

For **BYTE**, `C = A * B` and `C = A / B` are legal when `A`, `B`, and `C` are all `BYTE`. The transpiler lowers them to visible `ADD` / `SUB` loops in the generated ASM.

For `BCD(n)`, only `+` and `-` are defined. Decimal multiply and divide may be added in a future release as dedicated BCD routines.

In generated 4004 assembly

The 4004 provides native **add** and **subtract** on nibbles (`ADD`, `SUB`, with carry). It does **not** provide `MUL` or `DIV` instructions.

When source uses `*` or `/`, the transpiler **lowers** the operator to visible 4004 code:

Source operator	Typical generated pattern
`+`	`ADD` (and carry propagation for wider types)
`-`	`SUB` (and borrow propagation for wider types)
`*`	Repeated `ADD` (add-and-shift or counter-driven accumulation)
`/`	Repeated `SUB` (trial subtraction / quotient counter)

Wider types (`BYTE`) use the same lowering inside multi-nibble routines. The listing shows the loops; it does not call a black-box runtime that hides the algorithm.

How to read this document

- **"Supported in source"** means the operator may appear in QuadBasic expressions for that type.
- **"Lowered to … in ASM"** describes the implementation strategy. It does **not** mean the operator is unavailable in source.
- **"Not defined for `BCD(n)`"** applies only to decimal types in this specification, not to `NIBBLE` or `BYTE`.

NIBBLE

`NIBBLE` is the native data type of the 4004. It is what every working register, the accumulator, and every digit of RAM holds.

Definition

Property	Value
Bits	4
Range	0..15 (unsigned)
Default storage	CPU register R0..R9
Size in nibbles	1

Operations

`NIBBLE` supports all arithmetic, logical and comparison operators native to the 4004:

Category	Operators
Arithmetic	`+`, `-`, `*`, `/`, unary `-`
Bitwise	`AND`, `OR`, `XOR`, `NAND`, `NOR`, `XNOR`, `NOT`
Comparison	`=`, `<>`, `<`, `>`, `<=`, `>=`

In source: all operators in the table above are available in expressions (for example `C = A * B`, `C = A / B`).

In generated ASM: `+` and `-` map to `ADD` and `SUB` on the nibble. `*` and `/` lower to loops built from those same instructions (no `MUL` or `DIV` opcode).

Storage Behaviour

By default the compiler assigns each `NIBBLE` to a free CPU register starting at R0 and going up to R9. **R10..R15 are not available for user variables.** The compiler and the QuadBasic runtime in ROM page F use them as working registers during expression evaluation, RAM addressing (`FIM`/`SRC`), and library calls.

All ten user registers **R0 through R9** may hold `NIBBLE` variables. None of them is reserved for internal use.

If the program declares more `NIBBLE` variables than there are free registers, the compiler raises a compile-time error (see Errors). It does **not** silently spill to RAM. The program must either reduce the number of nibbles by reusing variables, or promote some of them to `BYTE` when values exceed 15.

This is deliberate: spilling to RAM without an explicit declaration would hide the cost of the variable and break the "what you write is what you see in ASM" guarantee.

Example

```
10 DIM A AS NIBBLE
20 DIM B AS NIBBLE
30 DIM C AS NIBBLE
40 A = 7
50 B = 3
60 C = A + B
70 PRINT C
80 END
```

Generated ASM for line 60 (illustrative):

```
; 60 C = A + B
LD  R0      ; ACC = A
CLC
ADD R1      ; ACC = A + B (mod 16)
XCH R2      ; C = ACC
```

The generated ASM contains a literal `ADD R1` instruction.

BYTE

`BYTE` is a two-nibble unsigned integer. It models how a 4004 programmer used **manual carry propagation** to handle values larger than 15.

Definition

Property	Value
Bits	8
Range	0..255 (unsigned)
Default storage	RAM 4002 (2 consecutive digits)
Size in nibbles	2

Layout

A `BYTE` occupies two consecutive nibble digits within the same RAM 4002 register. The high nibble is stored at the lower digit index, the low nibble at the higher digit index.

```
RAM[bank, chip, reg, d]   = HI nibble
RAM[bank, chip, reg, d+1] = LO nibble
```

This layout is documented in the Variable Map (see The Variable Map).

Working registers during evaluation

The stored value of a `BYTE` always remains in RAM. While a `BYTE` expression is evaluated or printed, the listing may load a working copy into:

Register	Nibble
R14	Low (0..15)
R15	High (0..15)

Decimal value = (high × 16) + low. Example: decimal 200 is high = 12, low = 8. After the statement finishes, inspect RAM (or the Variable Map), not R14/R15 alone.

Operations

Category	Operators	Notes
Arithmetic	`+`, `-`, `*`, `/`, unary `-`	Multi-nibble; carry between LO and HI
Comparison	`=`, `<>`, `<`, `>`, `<=`, `>=`	Compare HI first, then LO if equal
Logical	`AND`, `OR`, `XOR`, `NOT`	Nibble-by-nibble
Control flow	`FOR` / `NEXT`	Loop variable must be `BYTE`; range 0..255; STEP -255..255

In source: `C = A * B` and `C = A / B` are valid when `A`, `B`, and `C` are all `BYTE`.

In generated ASM: `+` and `-` use two-nibble carry or borrow between LO and HI. `*` and `/` lower to multi-nibble loops (shift-and-add or repeated subtract) built from `ADD` and `SUB`, not from hidden multiply or divide hardware.

Example

```
10 DIM A AS BYTE
20 DIM B AS BYTE
30 DIM C AS BYTE
40 A = 200
50 B = 87
60 C = A + B
70 END
```

Generated ASM for line 60 (illustrative, with Variable Map: A at `RAM[0,0,0,0..1]`, B at `RAM[0,0,0,2..3]`, C at `RAM[0,0,0,4..5]`):

```
; 60 C = A + B   (BYTE addition with manual carry)
LDM 0
DCL                     ; bank 0
; --- LO halves ---
FIM 0P, 001H            ; chip 0, reg 0, dig 1 (A.LO)
SRC 0P
RDM
XCH R10                 ; R10 = A.LO (system temp)
FIM 0P, 003H            ; B.LO
SRC 0P
RDM
CLC
ADD R10                 ; ACC = A.LO + B.LO  (CY = carry to high)
FIM 0P, 005H            ; C.LO
SRC 0P
WRM                     ; store C.LO, CY preserved
; --- HI halves ---
FIM 0P, 000H            ; A.HI
SRC 0P
RDM
XCH R10                 ; R10 = A.HI (system temp)
FIM 0P, 002H            ; B.HI
SRC 0P
RDM
ADD R10                 ; ACC = A.HI + B.HI + CY   (with carry from LO)
FIM 0P, 004H            ; C.HI
SRC 0P
WRM                     ; store C.HI
```

The generated ASM shows an explicit two-stage addition with carry propagation. This is the canonical 1971 way to add a byte on a 4004.

Twelve-bit addresses (no `WORD` type)

QuadBasic does **not** provide a 12-bit binary scalar. The 4004 still uses 12-bit ROM addresses: model them with **two nibbles** (e.g. `HI`, `LO`), `PEEK @ROM(HI, LO)`, `BYTE` for 0..255, `BCD(n)` for larger decimal literals, and explicit casts (`BYTE(x)`, `BCD4(x)`). Integer literals with **no** context larger than **255** are rejected unless you assign to a variable of sufficient type (for example `BCD(4)`).

BCD

`BCD(n)` is a Binary-Coded Decimal integer with **n decimal digits**, where n is a compile-time constant between 1 and 16. This is the type the Intel 4004 was originally designed for: the Busicom 141-PF calculator computed exclusively in BCD.

Definition

Property	Value
Decimal digits	n (1..16)
Bits	4n
Range	0..(10ⁿ − 1)
Default storage	RAM 4002 (n consecutive digits, same register)
Size in nibbles	n

Layout

A `BCD(n)` value occupies n consecutive nibble digits within the same RAM 4002 register. The most significant decimal digit is stored at the lowest digit index, the least significant at the highest index.

```
For BCD(4) at RAM[bank, chip, reg, d]:
  RAM[bank, chip, reg, d  ] = digit 3 (most significant)
  RAM[bank, chip, reg, d+1] = digit 2
  RAM[bank, chip, reg, d+2] = digit 1
  RAM[bank, chip, reg, d+3] = digit 0 (least significant)
```

Because a single 4002 register holds 16 nibbles, any `BCD(n)` with n ≤ 16 fits within one register. `BCD(17)` or larger is not allowed.

Operations

Category	Operators	Notes
Arithmetic	`+`, `-`	Per-digit, with `DAA` after each `ADD`
Comparison	`=`, `<>`, `<`, `>`, `<=`, `>=`	Compare most significant digit first

In source: only `+` and `-` are defined for `BCD(n)` in this specification. Bitwise operators and `*` and `/` are not part of the `BCD` operator set.

In generated ASM: `+` and `-` run per decimal digit with `ADD` followed by `DAA`. This section does **not** restrict `*` or `/` on `NIBBLE` or `BYTE`.

The Role of DAA

The 4004 has a dedicated `DAA` instruction (Decimal Adjust Accumulator). When the program adds two BCD digits with `ADD`, the binary result may be invalid BCD (for example 7 + 8 = 15 in binary, but 15 is not a valid BCD digit). `DAA` corrects this by adding 6 to the accumulator if needed, producing the correct BCD digit and an updated carry.

Without a `BCD` type, `DAA` rarely appears in generated code. With `BCD`, every addition explicitly emits `DAA`—one of the most distinctive instructions of the 4004.

Example

```
10 DIM TOTAL AS BCD(4)
20 DIM ITEM  AS BCD(4)
30 TOTAL = 1234
40 ITEM  = 5678
50 TOTAL = TOTAL + ITEM
60 END
```

Generated ASM for line 50 (illustrative, simplified, one digit shown):

```
; --- BCD(4) ADD: one digit (LSD), then repeat for d2, d1, d0 ---
LDM 0
DCL
FIM 0P, 003H            ; TOTAL.d0 (least significant)
SRC 0P
RDM
XCH R10                 ; R10 = TOTAL.d0 (system temp)
FIM 0P, 007H            ; ITEM.d0
SRC 0P
RDM
CLC
ADD R10                 ; ACC = sum (possibly > 9)
DAA                     ; ADJUST: produces valid BCD digit + CY
FIM 0P, 003H
SRC 0P
WRM                     ; store TOTAL.d0
; ... repeat for d1, d2, d3, propagating CY each time
```

The `DAA` instruction is plainly visible in the generated code, exactly where a hand-written 1971 BCD routine would have placed it.

Sizing Examples

Declaration	Bytes	Use Case
`BCD(2)`	1 nibble pair	Two-digit counters (00..99)
`BCD(4)`	2 nibble pairs	Scores, prices
`BCD(8)`	4 nibble pairs	Calculator-style values (00000000..99999999)
`BCD(16)`	1 full register	Maximum supported precision

Arrays

QuadBasic supports one-dimensional arrays of any scalar type.

Definition

Property	Value
Element type	`NIBBLE`, `BYTE`, or `BCD(n)`
Element count	Compile-time constant N (1..256)
Default storage	RAM 4002 (contiguous)
Size in nibbles	N × size(element type)

Declaration

```
DIM ARR(16) AS NIBBLE
DIM BUF(8)  AS BYTE
DIM TAB(4)  AS BCD(4)
```

The element count appears in parentheses **before** the `AS` keyword.

Indexing

Array elements are accessed with parentheses around an index expression:

```
ARR(3) = 5
X = ARR(I)
```

The index expression may be a literal, a variable, or a compound expression. The compiler emits `FIM`/`SRC` instructions to compute the runtime address from the index.

Layout in RAM

Each array is placed in a region of RAM 4002 chosen by the compiler. Elements are stored in increasing digit order within a register, and elements that do not fit in one register continue in the next register of the same chip.

Element 0 lives at the lowest address. For an array `ARR(N) AS T` placed at base `RAM[bank, chip, reg₀, dig₀]`:

```
Element 0 -> RAM[bank, chip, reg₀, dig₀]
Element 1 -> RAM[bank, chip, reg₀, dig₀ + size(T)]
...
```

The exact base address is reported in the Variable Map (see The Variable Map).

Indexing Cost

Indexing an array requires computing the runtime address. For an array of `NIBBLE`, this is a simple multiplication of the index by 1 (free) and an offset add. For an array of `BYTE`, it is a multiplication by 2. For an array of `BCD(n)`, it is a multiplication by n.

The generated code uses `FIM` with the base address followed by additive register manipulation to point `SRC` at the correct digit. Each step is visible in the ASM listing.

Bounds Checking

Array indexing is **not** bounds-checked at runtime. If the program writes `ARR(I)` with I out of range, the resulting memory access is whatever the compiler computes from the formula. This matches the philosophy of "what you write is what runs": adding bounds checks would emit hidden conditional jumps not requested by the source code.

Multi-Dimensional Arrays

Not supported. A 2D array can be simulated with a 1D array and explicit index arithmetic (`ARR(I*W + J)`).

Storage Model

The compiler assigns each declared variable to a concrete location: a CPU register, a region of RAM, or a fragment of RAM. The rules are deterministic and reported in the Variable Map.

Default Allocation Rules

Type	Default Location	Allocation Strategy
`NIBBLE`	CPU register	First free register in R0..R9, in declaration order (all ten may be used)
`BYTE`	RAM 4002	First free pair of consecutive digits in any chip; working copy in R14/R15 while active
`BCD(n)`	RAM 4002	First free run of n consecutive digits in any chip
Arrays	RAM 4002	Contiguous region in a dedicated chip when possible

Scalars (NIBBLE/BYTE/BCD) are packed into shared chips. Arrays prefer chips that are otherwise free, to keep the indexing arithmetic simple.

Reserved system RAM

QuadBasic reserves **32 RAM cells** on the **last RAM chip** (bank 3, chip 3): registers **2** and **3** (`RAM[3, 3, 2, 0..15]` and `RAM[3, 3, 3, 0..15]`). They never appear in the Variable Map.

Region	Owner	Purpose
`RAM[3, 3, 2, 0..15]`	Compiler	General work area (long conversions, nested expressions)
`RAM[3, 3, 3, 0]`	Compiler	Carry for multi-digit `BYTE` math
`RAM[3, 3, 3, 1..12]`	Compiler	Short temps, `BCD` conversion helpers, expression spill
`RAM[3, 3, 3, 13]`	Runtime	Current `PRINT` device id
`RAM[3, 3, 3, 14..15]`	Runtime	`RND` generator state

Registers 0 and 1 on that same chip remain available for your variables if a very large program needs them.

Allocation Exhaustion

If the program declares so many `NIBBLE` variables that R0..R9 are exhausted, the compiler raises `TOO_MANY_NIBBLE_VARIABLES`. The compiler does **not** silently move nibbles to RAM. Promote some variables to `BYTE` when wider storage is required.

If RAM is exhausted (very unusual; the addressable data model has **1,024** nibbles), the compiler raises `OUT_OF_RAM`.

The Variable Map

Every transpilation emits a **Variable Map** as a comment block near the top of the generated ASM file, immediately after the standard header. The Variable Map lists every declared variable and its assigned location, in declaration order.

The Variable Map is the single point of truth for where each variable lives. It documents the memory layout of the compiled program.

Format

```
; ================================================================
; BASIC4 VARIABLE MAP
; ================================================================
; #   Name       Type          Storage
; --- ---------- ------------- ----------------------------------------
; 0   A          NIBBLE        R0
; 1   B          NIBBLE        R1
; 2   COUNTER    BYTE          RAM[0,0,0,0..1]  (HI,LO)
; 3   SCORE      BCD(4)        RAM[0,0,0,2..5]  (d3,d2,d1,d0)
; 4   ARR        NIBBLE(16)    RAM[0,1,0..0,0..15]
; --- ---------- ------------- ----------------------------------------
; Registers used: R0..R1 of R0..R9            (8 free)
; RAM used:       6 nibbles in chip 0, 16 nibbles in chip 1
; ================================================================
```

Columns

Column	Meaning
`#`	Declaration order (0-based)
`Name`	Variable identifier as written in source
`Type`	Full type specification (`NIBBLE`, `BYTE`, `BCD(n)`, `T(N)` for arrays)
`Storage`	Canonical location with explicit nibble layout

Storage Notation

- `R0..R9` — single CPU register
- `RAM[b, c, r, d]` — single nibble at bank/chip/reg/digit
- `RAM[b, c, r, d₁..d₂]` — span of consecutive digits within a register
- `RAM[b, c, r₁..r₂, d₁..d₂]` — span across multiple registers (arrays only)

For multi-nibble scalars (`BYTE`, `BCD(n)`), the order of digits is annotated in parentheses (`(HI,LO)`, `(d3,d2,d1,d0)`).

Allocation Summary

The map ends with a short summary indicating how many registers are used out of R0..R9 and how many RAM nibbles are used per chip.

Cross-Reference

Every operation in the generated ASM that touches a variable includes a short comment with the variable name and its location, allowing the reader to navigate from any instruction back to the Variable Map without scrolling.

DIM declarations

Every variable must be declared with `DIM` before being used. There are no implicit declarations and no untyped variables.

Grammar

```
DimStatement   ::= "DIM" DimItem { "," DimItem }
DimItem        ::= Identifier [ "(" IntegerLiteral ")" ] "AS" TypeSpec
TypeSpec       ::= "NIBBLE" | "BYTE" | "BCD" "(" IntegerLiteral ")"
```

Examples

```
10 DIM A AS NIBBLE
20 DIM B AS BYTE
30 DIM C AS BCD(4)
40 DIM ARR(16) AS NIBBLE
```

Multiple Declarations per Statement

A single `DIM` may declare several variables of the **same type and storage class**:

```
10 DIM A, B, C AS NIBBLE
20 DIM X, Y    AS BYTE
```

Mixing types in one `DIM` statement is **not** allowed. The following is illegal:

```
10 DIM A AS NIBBLE, B AS BYTE        ' ERROR: type mismatch in DIM list
```

Position in the Program

All `DIM` statements must appear **before any executable statement**. The program begins with a header section that may contain only:

- `REM` and `'` comments
- `DIM` declarations
- `CONST` definitions

The first non-comment, non-`DIM`, non-`CONST` statement closes the header. Any `DIM` that appears after this point is a compile-time error (`DIM_AFTER_CODE`).

No Initialiser

`DIM` does **not** accept an initialisation value. Initial values must be assigned with a normal assignment statement after declaration:

```
10 DIM A AS NIBBLE
20 DIM B AS BYTE
30 A = 5             ' explicit assignment
40 B = 100           ' explicit assignment
```

This rule keeps every assignment visible in the ASM output and makes the cost of initialisation impossible to overlook.

Initialization Rules

User Variables

User variables are **not** automatically zero-initialised by the compiler. After power-on, registers and RAM hold whatever the hardware happens to have. Assign a value to every variable before reading it.

Reading an uninitialised variable is not a compile error (the compiler cannot in general detect it) but produces undefined behaviour.

Auto-Zero of RAM

When the program contains at least one `BYTE`, `BCD` or array variable, the compiler emits a short **boot prologue** that zero-fills every RAM nibble assigned to such variables. The prologue runs once, before the first user instruction.

The prologue is reported in the Variable Map summary and appears as a labelled block in the generated ASM:

```
; --- BOOT: zero variable RAM ---
B4_INIT:
    LDM 0
    DCL
    ...
```

`NIBBLE` variables in CPU registers are **not** zero-initialised by the prologue. Register initialisation is left to user code because each register is touched explicitly.

Runtime State

The runtime state regions (`RAM[3,3,3,13..15]`) are initialised by the QuadBasic runtime as needed (for example, the PRNG seeds itself on first call). User programs should not depend on the initial values of runtime state.

CONST

`CONST` defines a compile-time constant with optional type annotation.

Grammar

```
ConstStatement ::= "CONST" Identifier [ "AS" TypeSpec ] "=" LiteralExpression
```

Examples

```
10 CONST PI       = 3                    ' inferred NIBBLE
20 CONST LIMIT    AS BYTE  = 200
30 CONST PATTERN  AS NIBBLE = $A         ' hex literal
40 CONST MASK     AS BYTE  = %11001010   ' binary literal
50 CONST PRICE    AS BCD(4) = 1995
```

Semantics

A `CONST` is a name for a literal value. It generates **no runtime storage** and **no runtime instructions**. Every reference to a `CONST` is replaced by the literal value at compile time. Constants therefore consume no CPU registers and no RAM.

Type Rules

- If the type is omitted, the compiler infers the smallest type that fits the literal: `NIBBLE` for 0..15, `BYTE` for 16..255. Literals **above 255** must use an explicit context (for example assignment to `BCD(n)` or `CONST K AS BCD(4) = …`); bare literals `256..` are rejected in expressions without a destination type.
- BCD literals must have an explicit `AS BCD(n)` annotation.
- Literals can be expressed in decimal, hexadecimal (`$` prefix) or binary (`%` prefix).

Use in Expressions

`CONST` names are treated identically to literal values in expressions. They participate in type checking as their declared type.

```
10 CONST GREEN = 3
20 DIM COLOR AS NIBBLE
30 COLOR = GREEN     ' equivalent to COLOR = 3
```

Position

`CONST` statements live in the program header alongside `DIM` (see DIM Syntax). Both may be intermixed within the header.

Type Compatibility and Casts

QuadBasic is **strictly typed**. There are no implicit conversions between distinct types in expressions. Every cross-type operation requires an explicit cast.

Assignment Rules

An assignment `dst = src` is legal only when one of these conditions holds:

1. **Same type.** `dst` and `src` have identical type, including identical BCD width.
2. **Widening within the same family.** `BYTE := NIBBLE` is allowed. The source value is zero-extended.
3. **Explicit cast.** The source expression is wrapped in a cast.

All other combinations are compile errors. Specifically:

- `NIBBLE := BYTE` is **not** an implicit narrowing; it requires `NIBBLE(byte_var)`.
- `BCD(n) := BCD(m)` with `n ≠ m` is **not** allowed; use a cast.
- `BCD(n) := BYTE` and `BYTE := BCD(n)` are **never** implicit. They require explicit casts because BCD and binary are different encodings of the integer value.

Expression Type Rules

In an expression, operands of a binary operator must have identical type. There is no automatic promotion in expressions:

```
10 DIM A AS NIBBLE
20 DIM B AS BYTE
30 C = A + B          ' ERROR: type mismatch
40 C = BYTE(A) + B    ' OK
```

This rule is strict on purpose. Implicit promotion would generate hidden conversion code and silently change the size of arithmetic.

Cast Operators

QuadBasic provides one cast per scalar type:

Cast	Argument	Result	Notes
`NIBBLE(x)`	Any scalar	NIBBLE	Truncates to the low nibble
`BYTE(x)`	Any scalar	BYTE	Zero-extends from narrower, truncates from wider
`BCD2(x)` to `BCD16(x)`	Binary scalar	BCD(n)	Converts binary to BCD via **inline** mainline loops (compiler RAM scratch), not page F

The BCD casts (`BCDn(x)` and `NIBBLE/BYTE(bcd_x)`) generate **explicit conversion sequences** in the ASM output. The cost of switching encodings is visible in the listing.

BCD → binary (`BYTE`) value range

For operands of type `BCD(n)`, the runtime converts packed decimal digits to **unsigned binary** (ROM `B4_BCD2BIN`).

- `BYTE(bcd)` — result is an 8-bit value. If the decimal quantity in the BCD variable exceeds **255**, the stored `BYTE` is only the low eight bits (same as assigning a wider binary value to `BYTE`); use BCD decimals **0..255** for a predictable display value.

Twelve-bit ROM addresses are modelled with **two nibbles** (`HI`, `LO`) and `PEEK @ROM(HI, LO)`, not with a dedicated 12-bit scalar type.

There is **no runtime error** when the decimal is out of range for the binary width; the result truncates / wraps to the target width.

What Cannot Be Cast

- Cast from non-integer expressions: not allowed (there are no non-integer types).
- Cast on arrays: arrays cannot be cast as a whole; cast individual elements.

Errors

The following compile-time errors are defined by the typed variable model. Each error has a stable identifier and a human-readable message.

Code	Identifier	Message
E101	DIM_AFTER_CODE	DIM must appear before any executable statement
E102	UNDECLARED_VARIABLE	Variable '<NAME>' is not declared. Add a DIM statement at the top of the program
E103	TYPE_MISMATCH	Type mismatch in expression. Left side is <T1>, right side is <T2>
E104	NARROWING_REQUIRES_CAST	Assigning <SRC_TYPE> to <DST_TYPE> narrows the value. Use an explicit cast
E105	BCD_BINARY_MIX	BCD and binary types cannot be mixed. Use BCDn(...) or NIBBLE/BYTE(...) to convert
E106	TOO_MANY_NIBBLE_VARIABLES	Too many NIBBLE variables in registers (R0..R9 exhausted). Move some to RAM or use BYTE
E107	REG_ALREADY_USED	Register R<N> is already assigned to '<OTHER_NAME>'
E108	RAM_ALREADY_USED	RAM nibble RAM[<B>,<C>,<R>,<D>] is already assigned to '<OTHER_NAME>'
E109	RAM_RESERVED	Address is in reserved system RAM (last chip, registers 2–3). Pick a different location
E110	OUT_OF_RAM	Out of RAM 4002. Reduce the number or size of variables
E111	BCD_OUT_OF_RANGE	BCD(<N>) is invalid. N must be between 1 and 16
E112	ARRAY_SIZE_OUT_OF_RANGE	Array size must be between 1 and 256
E113	MIXED_TYPES_IN_DIM_LIST	All variables in a single DIM statement must share the same type
E114	LITERAL_OUT_OF_RANGE	Literal <VAL> does not fit in <TYPE>
E115	INVALID_OPERATOR_FOR_TYPE	Operator '<OP>' is not defined for type <TYPE>

Existing errors not listed here keep their previous identifiers.

Type system examples

Example 1: NIBBLE Counter

```
10 DIM I AS NIBBLE
20 I = 0
30 PRINT I
40 I = I + 1
50 IF I < 10 THEN GOTO 30
60 END
```

The Variable Map shows `I -> R0`. Lines 30 and 40 expand to direct `ADD`/`XCH` instructions on R0.

Example 2: BYTE Score

```
10 DIM SCORE AS BYTE
20 DIM DELTA AS BYTE
30 SCORE = 100
40 DELTA = 25
50 SCORE = SCORE + DELTA
60 END
```

The Variable Map shows `SCORE -> RAM[0,0,0,0..1]`, `DELTA -> RAM[0,0,0,2..3]`. Line 50 generates a full byte addition with manual carry between two nibbles.

Example 3: BCD Calculator

```
10 DIM A AS BCD(4)
20 DIM B AS BCD(4)
30 DIM C AS BCD(4)
40 A = 1234
50 B = 5678
60 C = A + B
70 END
```

Line 60 generates four `ADD` + `DAA` sequences, one per decimal digit, propagating carry through the digits in increasing significance.

Example 4: Array Sum

```
10 DIM ARR(8) AS NIBBLE
20 DIM I AS NIBBLE
30 DIM TOTAL AS BYTE
40 TOTAL = 0
50 FOR I = 0 TO 7
60   TOTAL = TOTAL + BYTE(ARR(I))
70 NEXT I
80 END
```

The cast `BYTE(ARR(I))` is required because `ARR(I)` is a NIBBLE and `TOTAL` is a BYTE. The cast is explicit in source and in the generated ASM.

Example 5: ROM address as HI / LO

```
10 DIM HI AS NIBBLE
20 DIM LO AS NIBBLE
30 HI = 3
40 LO = 8
50 ' use PEEK @ROM(HI, LO) to read a cell; 12-bit paths stay explicit
60 END
```

Twelve-bit addresses are split across two nibbles; the compiler does not provide a single 12-bit integer variable.

Example 6: Constants and Mixed Casts

```
10 CONST FULL AS BYTE = 255
20 DIM A AS BYTE
30 DIM B AS NIBBLE
40 A = FULL
50 B = NIBBLE(A)        ' explicit narrowing
60 END
```

Without the explicit `NIBBLE(A)`, line 50 would raise `NARROWING_REQUIRES_CAST`.

Out of Scope

The following features are deliberately not part of the type system. The reasons are recorded here so they are not revisited without explicit decision.

`BIT` Type

The 4004 has no bit-test, bit-set or bit-clear instructions. Bits are manipulated only through nibble-wide logical operations on `NIBBLE` values. A `BIT` type would be either decorative (mapping to a single-bit slice of a nibble) or simulated (emitting hidden mask-and-shift code), and neither matches the design goals of this type system.

Signed Integers

The 4004 has no signed comparison instructions and no sign extension. Programs that need negative values must implement two's-complement arithmetic explicitly, exactly as a 1971 programmer would.

Floating Point

Out of scope by the historical premise of the project. Floats did not exist on the 4004 and adding them would require either software emulation of substantial cost or a co-processor that did not exist.

Strings

QuadBasic supports string **literals** inside `PRINT` statements. It does not support string variables, string concatenation, string slicing, or string comparison. These would require either dynamic memory (which the 4004 does not provide) or fixed-size buffer machinery that distracts from the type system.

`PRINT`: `CHR`, `DEC`, and `HEX`

These forms are **only** valid as `PRINT` items (not as general expressions).

- `CHR(lo, hi)` — emits one ASCII byte: `lo` and `hi` are the low and high nibbles of the code unit (same encoding as elsewhere in QuadBasic).
- `DEC(expr)` — prints `expr` in **decimal**, explicitly. It is equivalent to printing a numeric expression without `DEC`, and accepts the same scalar types as ordinary numeric `PRINT` (including `BCD`).
- `HEX(expr)` — prints **hexadecimal digits** (`0`–`9`, `A`–`F`) without a `0x` prefix. Operand must be a scalar `NIBBLE` or `BYTE` (not `BCD`). For `BYTE`, the **high nibble is printed first**, then the low nibble (big-endian nibbles). For `NIBBLE`, one digit is printed.

Pointers and Pointer Arithmetic

Variables in QuadBasic refer to a fixed location chosen at compile time. There is no `&A` operator and no `*P` dereference. Indirect access is available through `PEEK`/`POKE` with computed indices, which is the 1971-faithful way to do indirection on a 4004.

Multi-Dimensional Arrays

Only one-dimensional arrays are supported. Two-dimensional patterns can be flattened by hand: `ARR(I * WIDTH + J)`.

Implicit Promotion in Expressions

Every binary operator requires operands of identical type. Implicit promotion would hide the cost of conversions and contradicts the type-strictness rule.

User-Defined Types / Structs

Not supported. Group variables by naming convention if structure is needed.

CPU Registers and System RAM

The Intel 4004 has sixteen 4-bit registers, **R0** through **R15**. In a QuadBasic program, these registers play different roles depending on how your variables are declared.

- `NIBBLE` **variables** are stored in **R0..R9**. Their values remain in those registers for the whole program unless your source code assigns a new value.
- **Wider types** (`BYTE`, `BCD`, arrays) are stored in **RAM 4002**. The CPU registers hold only temporary copies while an expression is evaluated or while output is generated.
- **R10..R15** are not available for user variables. The compiler and the QuadBasic runtime in ROM page F use them as working registers during instruction sequences.

The register layout below describes so you can read the generated assembly, use the Variable Map, and interpret the register window in the simulator without guessing which registers belong to your program.

Type sizes, RAM allocation, and `DIM` rules are part of the QuadBasic type system (`NIBBLE`, `BYTE`, `BCD(n)`, arrays).

Register Regions

Region	Registers	Holds	Persists between statements?
User	**R0..R9**	Declared `NIBBLE` variables	Yes, for the lifetime of the program
System	**R10..R15**	Expression temporaries, addressing, runtime helpers	No — treat as scratch

```
R0  R1  R2  R3  R4  R5  R6  R7  R8  R9 | R10 R11 R12 R13 R14 R15
|<-------- user NIBBLE variables -------->|<----- system / runtime ----->|
```

If you need more than ten independent values in the 0..15 range at once, declare some variables as `BYTE` (stored in RAM) instead of adding more `NIBBLE` names.

User Registers R0 through R9

Assignment

Each `DIM name AS NIBBLE` receives the next free register, starting at **R0**, in the order of `DIM` statements in the program header.

Example:

```
DIM A AS NIBBLE      ' R0
DIM B, C AS NIBBLE   ' R1, R2
```

The **Variable Map** at the top of the generated ASM lists the exact register for each name.

Access in generated ASM

Typical patterns:

```
LD  R0      ; load NIBBLE variable A into accumulator
...
XCH R0      ; store accumulator into A
```

Limits

Limit	Value
Maximum `NIBBLE` variables	10 (R0..R9)
Eleventh `NIBBLE`	Compile error `TOO_MANY_NIBBLE_VARIABLES`

The compiler does **not** place `NIBBLE` variables in RAM automatically. Use `BYTE` when you need more named values or ranges beyond 0..15.

What you can rely on

While your program runs (until you assign to that variable again), the value of a `NIBBLE` in **Rn** is the value of that variable. Inspecting **R0..R9** in the simulator is a direct way to watch `NIBBLE` data.

System Registers R10 through R15

These registers are **not listed** in the Variable Map as user storage. Any value you see in R10..R15 during a single line of ASM is almost certainly a short-lived intermediate result.

Register	Typical role in generated programs
R10	Temporary operand; logical operation input A; low nibble sent to the printer interface
R11	Temporary operand; logical operation input B; high nibble sent to the printer interface
R12	Address component for `FIM` / `SRC` (RAM and ROM access); decimal digit quotient during `PRINT`
R13	Digit index for `FIM` / `SRC`; print-started flag and scratch during decimal output
R14	Low nibble of an active `BYTE` while it is being processed
R15	High nibble of an active `BYTE` while it is being processed

Addressing (R12 and R13)

RAM and ROM accesses in MCS-4 code use the register pair loaded by `FIM` and selected by `SRC`. That mechanism uses **R12** and **R13**. While the listing shows `FIM` / `SRC`, those two registers should not be interpreted as program variables.

Do not use R10..R15 as variables

QuadBasic does not provide a `DIM` type mapped to R10..R15. If you need another named value, declare a `NIBBLE` (if it fits in R0..R9) or a `BYTE` in RAM.

Where Each Type Lives

Type	Primary storage	Registers during operations
NIBBLE	R0..R9 (one variable per register)	Same register as the variable
BYTE	RAM 4002 (2 nibbles)	R14 (low), R15 (high) while the value is active
BCD(n)	RAM 4002 (n nibbles)	System temporaries; see ASM listing
Arrays	RAM 4002	R12/R13 for computed addresses; element in accumulator

The canonical value of a `BYTE` is always in RAM. After a statement completes, check RAM (or the Variable Map), not R14/R15 alone, to see the stored result.

BYTE layout in RAM and in registers

In RAM, a `BYTE` uses two consecutive digits: **high nibble at the lower digit index**, **low nibble at the higher digit index**.

While a `BYTE` expression is evaluated, the working copy often appears as:

- R14 — low nibble (0..15)
- R15 — high nibble (0..15)

Decimal value = (high × 16) + low. Example: decimal 200 → high = 12, low = 8.

Reading the Generated ASM Header

Below the title comment, transpiled programs include a short register summary, for example:

```
; R0..R9   : declared NIBBLE variables (see Variable Map).
; R10 (T0) : expression temporaries, logical operands, subroutine results.
; R11 (T1) : expression temporaries, logical operand B, print data.
; R12 (ADR): FIM/SRC address; decimal quotient during PRINT.
; R13 (DIG): FIM digit index; print-started flag; PRINTDEC scratch.
; R14 (WLO): low nibble of active BYTE in RAM.
; R15 (WHI): high nibble of active BYTE in RAM.
; BYTE values live in RAM; R14/R15 are a per-statement working window.
```

Use this block together with the **Variable Map**: the map names your variables; the header explains registers that will change frequently but are not variables.

The Variable Map and Registers

The **Variable Map** lists only **user** storage:

- `NIBBLE` → `R0`, `R1`, …
- `BYTE` / `BCD` / arrays → `RAM[bank, chip, register, digits…]`

It ends with a summary such as:

```
; Registers used: R0..R3 of R0..R9            (6 free)
; RAM used: 12 nibbles
```

Registers used counts how many of **R0..R9** hold `NIBBLE` variables. **Free** registers are still available for more `NIBBLE` declarations if you edit and retranspile the program.

`NIBBLE` lines in the map are the only register assignments that belong to your source program.

Runtime Library and Registers

QuadBasic links selected operations to small routines in **ROM page F** (`0F00H`–`0FFFH`). Page 0 holds **stubs** (`B4_PRINTDEC:` … `JUN B4_PRINTDEC_F`) so generated code can `JMS` a short local name. Only routines your program actually uses are emitted; the combined page-F image must fit in **256 bytes** or transpilation fails.

Category	Page-F routine(s)	When linked	Registers / RAM
Logical	`B4_NOT` … `B4_XNOR`	Matching operator in source	**R10** = A, **R11** = B, result in **R10**
Random	`B4_RND` (also links `B4_XOR`)	`RND()`	Result in **R10**; state in `RAM[3, 3, 3, 14..15]`
Print	`B4_PRINTDEC`, `B4_PRINTCHAR`, `B4_PRINTHEXNYB`, `B4_PRINTLF`, `B4_SETDEV`	`PRINT`, `CLS`, strings / `CHR`	`CHR` and string bytes: **R10** = LO, **R11** = HI before `B4_PRINTCHAR`; full `BYTE` decimal print often uses mainline loops calling `B4_PRINTDEC` with **R14**/**R15**
BCD decode	`B4_BCD2BIN`	`NIBBLE(bcd)` / `BYTE(bcd)` casts	BCD address in **R12**/**R13** (`FIM`/`SRC`); binary result in **R14**/**R15**

Not in page F today:

- **Binary → BCD** (`BCD2(x)` … `BCD16(x)`): repeated divide-by-10 loops in **mainline** ASM (compiler scratch on the last RAM chip).
- **STROBE** (`B4_STROBE_F`): present in the runtime source but **not linked**; character and newline output **inlines** the ROM0 strobe inside `B4_PRINTCHAR_F` / `B4_PRINTLF_F`.

User `NIBBLE` **registers R0..R9 are not parameters** to page-F routines and are preserved across `JMS`. After any runtime call, do not assume **R10..R15** still hold a meaningful value.

Full stub catalog, calling conventions, and emission order: Appendix: QuadBasic Runtime Routines.

System RAM on the last chip

Your program's variables are listed in the **Variable Map** at the top of the generated ASM file. QuadBasic also uses a fixed area of **RAM 4002** that **never** appears in that map.

The Workbench has **16 RAM chips**. Your variables are allocated starting from the first chips. In typical programs they never reach the last one.

System storage is placed on the **last RAM chip** only:

- **Bank 3, chip 3** (the 16th chip on the board)
- **Registers 2 and 3** on that chip — **32 four-bit cells** in total

In map notation:

- `RAM[3, 3, 2, 0]` … `RAM[3, 3, 2, 15]` — compiler work area (16 cells)
- `RAM[3, 3, 3, 0]` … `RAM[3, 3, 3, 15]` — runtime state and short compiler helpers (16 cells)

Registers 0 and 1 on the same chip (another 32 cells) are **not** reserved. The compiler may assign your `BYTE`, `BCD`, or array variables there if the program needs a lot of RAM. That is unusual in small programs.

```
Last RAM chip (bank 3, chip 3) — 64 cells total

  Register 0   [16 cells]   Available for your variables (if needed)
  Register 1   [16 cells]   Available for your variables (if needed)
  Register 2   [16 cells]   QuadBasic compiler work area (reserved)
  Register 3   [16 cells]   Runtime + compiler helpers (reserved)
```

Out of **1,024** data cells in the whole RAM model, **32** are reserved for QuadBasic (~3%). The rest is available for your declarations.

Register 3 — fixed roles

These cells keep their meaning for the whole run of the program (except where noted as "only during one statement").

Cell `RAM[3, 3, 3, d]`	Used by	Purpose
0	Compiler	Carry flag (0 or 1) for multi-digit `BYTE` add/subtract
1 – 3	Compiler	Short temporary storage during expressions
4 – 6	Compiler	Temporary storage during binary-to-`BCD` conversion
7 – 8	Compiler	Scratch during wide expression lowering (legacy FIM pair save)
9 – 11	Compiler	Multiply/divide counter spill for **BYTE** only (`*` / `/` loops)
12	Compiler	Scratch during some print/lowering paths
13	Runtime	Active output device for `PRINT` (screen, printer, etc.)
14 – 15	Runtime	Internal state for `RND`

Cells **13 – 15** are updated by `PRINT` / device selection and `RND`. Do not rely on their values in your own logic.

Cells **0 – 12** may change during a line that performs wide arithmetic, a conversion, or a complicated expression. After that line finishes, you can ignore them unless you are stepping through the ASM listing.

Register 2 — compiler work area

All 16 cells in `RAM[3, 3, 2, 0]` … `RAM[3, 3, 2, 15]` are reserved for the **compiler only**.

They are used for heavier work that does not fit in the CPU's temporary registers, for example:

- Long binary-to-`BCD` conversions
- Intermediate results during division or multiplication
- Extra storage when an expression is deeply nested

This area has **no fixed cell-by-cell meaning** for you as the programmer. Treat it as scratch memory that QuadBasic may overwrite at any time.

Important: Compiler work in register 2 is designed so it does **not** overwrite runtime cells **13 – 15** in register 3. Your `RND` state and active `PRINT` device remain in register 3.

What you see in the simulator

Data type	Where to look
`NIBBLE` variables	Registers **R0 – R9** (see Variable Map)
`BYTE`, `BCD`, arrays	RAM addresses in the **Variable Map**
System area	**Not** in the Variable Map — last chip, registers 2 and 3

While a single line runs, a `BYTE` value may also appear briefly in **R14 – R15** as a working copy. **R12** and **R13** are also used for `FIM`/`SRC` addressing during that line. That is normal; the stored value is still in RAM at the address shown in the map.

Registers **R10 – R15** are also used as fast temporaries. They are not variables and are not listed in the map.

Rules for program authors

Do

- Use the **Variable Map** and **R0 – R9** as the source of truth for your data.
- Expect `RND` and `PRINT` routing to work without you managing cells 13 – 15.
- Ignore register 2 and most of register 3 unless you are studying generated code or debugging a conversion.

Do not

- Declare variables on `RAM[3, 3, 2, *]` or `RAM[3, 3, 3, *]` — the compiler will not place them there, and hand-written ASM must not use these cells for your own data.
- Assume a `BYTE` "lives" in R14 because it appeared there during `PRINT` — after the statement, only RAM counts.
- Read or write cells 13 – 15 in your own code to "remember" things — they belong to the runtime.

`BCD` and this RAM area

- `BCD` **addition and subtraction** use the CPU's decimal adjust (`DAA`) instruction digit by digit. Your `BCD` variables stay at the RAM addresses in the Variable Map.
- **Converting** between binary (`BYTE`) and `BCD` is more work. The compiler uses register **2** and parts of register **3** (cells 0 – 12) during that conversion. That cost is visible in the ASM listing.
- Runtime cells **13 – 15** are **not** used as scratch for those conversions.

Relation to the rest of the board

Bank 3 is also used by the **hardware simulation** for other features (for example, shift-register chains on a **different** chip in the same bank). That is separate from this QuadBasic system area.

QuadBasic's reserved block is only:

`RAM[3, 3, 2, 0..15]` **and** `RAM[3, 3, 3, 0..15]`

Quick reference

Item	Value
Total system cells	32
Location	Last RAM chip, registers 2 and 3
In Variable Map?	No
Your variables on same chip?	Yes, in registers 0 and 1 if the program needs that much RAM
`RND` / `PRINT` device	`RAM[3, 3, 3, 13..15]`
Compiler carry for `BYTE`	`RAM[3, 3, 3, 0]`

Inspecting Registers in the Simulator

When debugging a QuadBasic program:

1. Open the generated ASM and read the **Variable Map**.
2. For each `NIBBLE`, watch the assigned **R0..R9** register.
3. For each `BYTE`, watch the RAM nibbles listed in the map; use **R14**/**R15** only while stepping through the line that computes or prints that value.
4. Treat **R10..R13** as short-lived unless you are studying how `FIM`, `PRINT`, or runtime calls work.
5. Do not use **RAM[3, 3, 2, *]** or **RAM[3, 3, 3, *]** as program storage — see System RAM on the last chip.

A common mistake is to assume a `BYTE` variable "lives" in R14 because it appeared there during `PRINT`. After the statement finishes, the stored value is only in RAM.

Practical Limits

Situation	Recommendation
Up to 10 counters/flags in 0..15	Use `NIBBLE` in R0..R9
More than 10 such values	Use `BYTE` in RAM or reuse variables
Values 0..255	Use `BYTE`
Twelve-bit ROM addresses	Two nibbles (`HI` / `LO`) and `PEEK @ROM(HI, LO)` (no `WORD` type)
Decimal display (`BCD`)	Use `BCD(n)` in RAM
Watching data in the simulator	Map → `NIBBLE` in Rn, wider types in RAM

Keywords

Flow Control

Keyword	Description	Syntax
IF	Evaluates a conditional expression	`IF condition THEN ...`
THEN	Separator in IF structures	`IF X = 5 THEN ...`
ENDIF	Closes an IF block	`ENDIF`
END	Stops program execution immediately	`END`
GOTO	Unconditional jump to a line	`GOTO 100`
GOSUB	Subroutine call	`GOSUB 200`
RETURN	Return from subroutine	`RETURN`

Loops

Keyword	Description	Syntax
FOR	Begins a counted iteration loop	`FOR X = 1 TO 10`
TO	Specifies the upper limit of the loop	`FOR X = 1 TO 10`
STEP	Specifies the loop increment (optional)	`FOR X = 1 TO 10 STEP 2`
NEXT	Advances or terminates a counted iteration	`NEXT` or `NEXT X`

Declarations and Comments

Keyword	Description	Syntax
DIM	Declares typed variables (header only)	`DIM A AS NIBBLE` or `DIM A, B AS BYTE`
CONST	Named compile-time constant	`CONST NAME [AS type] = value`
LET	Explicit assignment (optional)	`LET A = 5` or `A = 5`
REM	Comment	`REM This is a comment`

Note: Comments can also use the `'` (apostrophe) character.

Output and Printing

Keyword	Description	Syntax
PRINT	Prints text or values	`PRINT "Hello"` or `PRINT A`
CHR	Function to print characters (only inside PRINT)	`PRINT CHR(lo, hi)`
CLS	Clears the screen	`CLS`

Memory Access

Keyword	Description	Syntax
PEEK	Reads from memory or ports	`PEEK @RAM(bank, chip, reg, digit), VAR`
POKE	Writes to memory or ports	`POKE @RAM(bank, chip, reg, digit), value`

Logical Operators (in Expressions)

Keyword	Description	Usage
NOT	Logical negation (unary)	`NOT X`
AND	Logical AND (binary)	`X AND Y`
OR	Logical OR (binary)	`X OR Y`
XOR	Exclusive OR (binary)	`X XOR Y`
NAND	Not AND (binary)	`X NAND Y`
NOR	Not OR (binary)	`X NOR Y`
XNOR	Not exclusive OR (binary)	`X XNOR Y`

Functions

Keyword	Description	Syntax
RND	Generates a random number (0-15)	`RND()`

Operators

Arithmetic Operators

Operator	Description	Example	Precedence
`+`	Addition	`A + B`	Medium
`-`	Subtraction	`A - B`	Medium
`*`	Multiplication	`A * B`	High
`/`	Division	`A / B`	High
`-` (unary)	Negation	`-5`	Very High

Arithmetic in source and in ASM: Full rules per type are under Arithmetic in Source and in ASM in Type System and Storage. In brief: `NIBBLE` and `BYTE` support `+`, `-`, `*`, `/` in source; the 4004 has no `MUL` / `DIV` opcodes, so `*` and `/` lower to `ADD` / `SUB` loops in the listing.

Comparison Operators

Operator	Description	Example
`=`	Equal to	`A = 5`
`<>`	Not equal to	`A <> 5`
`<`	Less than	`A < 5`
`>`	Greater than	`A > 5`
`<=`	Less than or equal to	`A <= 5`
`>=`	Greater than or equal to	`A >= 5`

Logical Operators

Logical operators are used as keywords (see previous section):

- `AND`, `OR`, `XOR`, `NAND`, `NOR`, `XNOR` (binary)
- `NOT` (unary)

Operator precedence:

1. Parentheses `()`
2. Unary operators (`NOT`, `-`)
3. Multiplication and division (`*`, `/`)
4. Addition and subtraction (`+`, `-`)
5. Logical operators (`AND`, `OR`, `XOR`, `NAND`, `NOR`, `XNOR`)

Functions

RND()

RND generates a pseudo-random 4-bit value (0-15) produced by the QuadBasic runtime.

Syntax:

```
RND()
```

Example:

```
10 DIM X AS NIBBLE
20 X = RND()
30 PRINT X
```

Note: No arguments required, but can be used with empty parentheses: `RND()`.

Runtime: `JMS B4_RND` leaves the pseudo-random nibble in **R10** (0..15). Internal 8-bit state lives in `RAM[3, 3, 3, 14..15]` and is updated on every call. Linking `B4_RND` also pulls `B4_XOR` for the mixing step.

CHR()

Converts two 4-bit values (lo, hi) into a character for printing. **Can only be used inside PRINT**.

Syntax:

```
PRINT CHR(lo, hi)
```

Example:

```
10 PRINT CHR(5, 10)
```

Parameters:

- `lo`: Low nibble (0-15) → **R10** before `B4_PRINTCHAR`
- `hi`: High nibble (0-15) → **R11** before `B4_PRINTCHAR`

Runtime: Same register convention as string literals (`LO` in **R10**, `HI` in **R11**).

Statements

Assignment

Syntax:

```
[LET] variable = expression
```

Examples:

```
10 A = 5
20 LET B = A + 3
30 C = RND()
```

IF-THEN

QuadBasic supports two forms of IF:

Inline IF (single statement)

```
10 IF A = 5 THEN PRINT "Is five"
```

Block IF (multiple statements)

```
10 IF A = 5 THEN
20   PRINT "Is five"
30   A = 0
40 ENDIF
```

Complex conditions:

```
10 IF A = 5 AND B < 10 THEN
20   PRINT "Condition met"
30 ENDIF

10 IF NOT A = 5 OR B > 10 THEN
20   PRINT "Other condition"
30 ENDIF
```

Nested IF:

```
10 IF A = 1 THEN
20 IF B = 2 THEN
30 C = 6
40 ENDIF
50 ENDIF
```

Limits:

- Maximum 8 comparisons in a single IF condition
- Nested IFs are supported: a block IF body may contain another inline or block IF; each block IF closes with its own ENDIF
- Inline IF after THEN accepts one statement; that statement may be another IF ... THEN ... on the same line
- Comparison operators can be combined with `AND` and `OR` within one condition

FOR-NEXT

Syntax:

```
FOR variable = start TO end [STEP increment]
  ...
NEXT [variable]
```

Examples:

```
10 FOR I = 1 TO 10
20   PRINT I
30 NEXT I

10 FOR X = 0 TO 15 STEP 2
20   PRINT X
30 NEXT

10 FOR J = 10 TO 0 STEP -1
20   PRINT J
30 NEXT J
```

Notes:

- `STEP` is optional (default is 1)
- `STEP` can be negative for descending loops
- Variable name in `NEXT` is optional

GOTO

Syntax:

```
GOTO line_number
```

Example:

```
10 A = 5
20 GOTO 50
30 PRINT "Not executed"
50 PRINT "Jumped here"
```

GOSUB / RETURN

Syntax:

```
GOSUB line_number
...
RETURN
```

Example:

```
10 GOSUB 100
20 PRINT "Returned from subroutine"
30 END

100 PRINT "In subroutine"
110 RETURN
```

PRINT

PRINT outputs one or more items to a selectable output target. Output targets are abstract devices defined by the QuadBasic runtime.

Complete syntax:

```
PRINT [@device[,]] item { ( ; | & ) item } [ ; | & ]
```

Output targets valid in QuadBasic source today:

- `@SCR` or unspecified: Screen (default)
- `@PRN`: Printer
- `@LCD`: LCD display
- `@DSK`: Disk
- `@NUL`: Null device (discards output)

Reserved for future extensions (not valid in source yet):

- `@COM`: Serial port (runtime device id 4)
- `@AUX`: Auxiliary port (runtime device id 5)

`PRINT @COM` and `PRINT @AUX` are **not** accepted by the transpiler today. The runtime reserves ids 4 and 5 for `B4_SETDEV`; source-level support will arrive in a future language extension.

PRINT items:

- Text strings: `PRINT "Hello world"`
- Expressions: `PRINT A` or `PRINT A + B` (decimal; `BYTE` uses multi-digit lowering in the listing)
- `PRINT CHR(lo, hi)` — one ASCII byte from two nibbles
- `PRINT DEC(expr)` — decimal explicitly (including `BCD`)
- `PRINT HEX(expr)` — hex digits for `NIBBLE` or `BYTE` only (`BYTE`: high nibble first)
- Multiple items: `PRINT "A="; A; " B="; B` (or use `&` instead of `;` between items)

Separators `;` and `&` (equivalent, only inside `PRINT`):

- Between items: concatenates on the same line
- At the end: does not print newline (`PRINT` with trailing `;` or `&` only)

Examples:

```
10 PRINT "Hello"
20 PRINT "A="; A
30 PRINT @PRN, "Printed"
40 PRINT "No newline";
50 PRINT "Continues on same line"
```

⚠️ IMPORTANT: PRINT Hardware Wiring (Fixed ROM Ports)

PRINT HARCODED ROM PORTS

The QuadBasic runtime uses **hardcoded ROM ports** for all PRINT operations. These ports are permanently assigned and cannot be changed:

- **ROM0.PORT bit0**: STROBE signal (0→1 edge) - automatically generated with every PRINT operation
- **ROM1.PORT**: High nibble (HI) of the byte being printed
- **ROM2.PORT**: Low nibble (LO) of the byte being printed

This means:

- Any external peripheral connected to ROM chips 0, 1, or 2 will conflict with PRINT operations
- When reading ROM ports 0, 1, or 2 with `PEEK @ROMPORT`, you may read values written by PRINT instead of your peripheral
- **Recommendation**: Use ROM chips 3 or higher (3-15) for external peripherals to avoid conflicts

How PRINT works internally:

1. Each character/byte is split into HI and LO nibbles
2. LO nibble is written to ROM2.PORT
3. HI nibble is written to ROM1.PORT
4. STROBE (0→1 edge) is generated on ROM0.PORT bit0
5. The output device reads the byte from ROM1/ROM2 and responds to the STROBE signal

Note: Even `PRINT ""` (empty string) executes `PRINTLF` which writes LF (0x0A) to ROM2.PORT, so ROM2 is always affected by PRINT operations.

CLS

Clears the screen and positions the cursor at the origin.

Syntax:

```
CLS
```

Example:

```
10 CLS
20 PRINT "Screen cleared"
```

DIM

Declares typed variables. All `DIM` statements belong in the program header (before executable statements). Initial values are assigned with separate assignment statements, not inside `DIM`.

Syntax:

```
DIM name AS type
DIM name1, name2 AS type
DIM arr(N) AS type
```

Examples:

```
10 DIM A AS NIBBLE
20 DIM B, C AS BYTE
30 DIM SCORE AS BCD(4)
40 DIM BUF(16) AS NIBBLE
```

Notes:

- Every variable must be declared before use
- Every variable needs a type; see Type System and Storage and DIM declarations for grammar, casts, and allocation

END

Terminates program execution.

Syntax:

```
END
```

Example:

```
10 IF A = 0 THEN END
20 PRINT "Continues"
```

REM / Comments

Syntax:

```
REM comment
' comment
```

Examples:

```
10 REM This is a comment
20 A = 5  ' Comment at end of line
30 ' Another comment
```

Expressions

Literals

- **Integers:** `5`, `0`, `15` (and wider literals where the target type allows)
- **Unsigned binary types:** `NIBBLE` 0..15, `BYTE` 0..255; twelve-bit ROM addresses use two nibbles plus `PEEK @ROM`
- **Strings:** `"Hello world"` (only in PRINT)

Variables

- Declared with `DIM` and a type before use
- Names: alphanumeric identifiers; not reserved words
- Case-insensitive

Operations

Arithmetic:

```
A + B
A - B
A * B
A / B
-A
```

Logical:

```
NOT A
A AND B
A OR B
A XOR B
A NAND B
A NOR B
A XNOR B
```

Comparisons (only in IF):

```
A = B
A <> B
A < B
A > B
A <= B
A >= B
```

Parentheses:

```
(A + B) * C
NOT (A AND B)
```

MCS-4 Memory Access

PEEK

Reads values from RAM, ROM memory or ports.

PEEK @RAM

Reads a nibble from 4002 RAM.

Syntax:

```
PEEK @RAM(bank, chip, reg, digit), destination_variable
```

Parameters:

- `bank`: RAM bank (0-7)
- `chip`: RAM chip (0-3)
- `reg`: Register (0-15)
- `digit`: Digit/nibble (0-3)
- `destination_variable`: Variable where the value is stored

Example:

```
10 PEEK @RAM(0, 1, 5, 2), X
20 PRINT X
```

PEEK @RAMSTATUS

Reads a status byte from 4002 RAM.

Syntax:

```
PEEK @RAMSTATUS(bank, chip, statusIndex), destination_variable
```

Parameters:

- `bank`: RAM bank (0-7)
- `chip`: RAM chip (0-3)
- `statusIndex`: Status index (0-3, must be a literal)
- `destination_variable`: Variable where the value is stored

Example:

```
10 PEEK @RAMSTATUS(0, 1, 0), STATUS
```

PEEK @ROM

Reads a complete byte from 4001 ROM.

Syntax:

```
PEEK @ROM(hi, lo), variable_hi, variable_lo
```

Parameters:

- `hi`: High nibble of address (0-15)
- `lo`: Low nibble of address (0-15)
- `variable_hi`: Variable for high nibble
- `variable_lo`: Variable for low nibble

Example:

```
10 PEEK @ROM(5, 10), HI, LO
20 PRINT "Byte:", HI, LO
```

PEEK @ROMPORT

Reads from the I/O port of a 4001 ROM chip.

Syntax:

```
PEEK @ROMPORT(chip), destination_variable
```

Parameters:

- `chip`: ROM chip (0-15)
- `destination_variable`: Variable where the value is stored

Example:

```
10 PEEK @ROMPORT(0), PORTVAL
```

Note: `PEEK @RAMPORT` is not supported (RAM port is write-only).

POKE

Writes values to RAM, ROM memory or ports.

POKE @RAM

Writes a nibble to 4002 RAM.

Syntax:

```
POKE @RAM(bank, chip, reg, digit), value
```

Parameters:

- `bank`: RAM bank (0-7)
- `chip`: RAM chip (0-3)
- `reg`: Register (0-15)
- `digit`: Digit/nibble (0-3)
- `value`: Value to write (expression)

Example:

```
10 POKE @RAM(0, 1, 5, 2), 10
20 A = 5
30 POKE @RAM(0, 1, 5, 3), A
```

POKE @RAMSTATUS

Writes a status byte to 4002 RAM.

Syntax:

```
POKE @RAMSTATUS(bank, chip, statusIndex), value
```

Parameters:

- `bank`: RAM bank (0-7)
- `chip`: RAM chip (0-3)
- `statusIndex`: Status index (0-3, must be a literal)
- `value`: Value to write (expression)

Example:

```
10 POKE @RAMSTATUS(0, 1, 0), 15
```

POKE @RAMPORT

Writes to the I/O port of a 4002 RAM chip.

Syntax:

```
POKE @RAMPORT(bank, chip), value
```

Parameters:

- `bank`: RAM bank (0-7)
- `chip`: RAM chip (0-3)
- `value`: Value to write (expression)

Example:

```
10 POKE @RAMPORT(0, 1), 12
```

POKE @ROMPORT

Writes to the I/O port of a 4001 ROM chip.

Syntax:

```
POKE @ROMPORT(chip), value
```

Parameters:

- `chip`: ROM chip (0-15)
- `value`: Value to write (expression)

Example:

```
10 POKE @ROMPORT(0), 8
```

VDP statements

High-level helpers for the VDP peripheral. They transpile to `POKE` / `PEEK` ROM-port traffic on the default wiring (ROM 7 = DATA, ROM 8 = CTRL, ROM 9 = STATUS).

The current implementation assumes the default playground wiring:

- ROM 8 = VDP CTRL.
- ROM 9 = VDP STATUS (VBLANK).

VDP PRESENT

Shows the completed back buffer on the visible display. Equivalent to `POKE @ROMPORT(8), 15`.

```
10 VDP PRESENT
```

VDP WAIT VBLANK

Waits until the VBLANK status bit is set. Equivalent to polling `PEEK @ROMPORT(9), V` until `V` is non-zero.

```
10 VDP WAIT VBLANK
```

VDP FRAME

Presents the current frame and then waits for VBLANK. Equivalent to `VDP PRESENT` followed by `VDP WAIT VBLANK`.

```
10 VDP FRAME
```

VDP MODE BITMAP

Enters bitmap mode. Equivalent to `POKE @ROMPORT(8), 11` followed by `POKE @ROMPORT(7), 0` (plus `4` if sprites are enabled).

```
10 VDP MODE BITMAP
```

VDP MODE TILE

Enters tile mode. Equivalent to `POKE @ROMPORT(8), 11` followed by `POKE @ROMPORT(7), 1` (plus `4` if sprites are enabled).

```
10 VDP MODE TILE
```

VDP MODE TEXT

Enters text mode with foreground and background palette indexes. Equivalent to command 11 with mode value `2`, then two DATA nibbles for `fg` and `bg` (plus `4` if sprites are enabled).

```
10 VDP MODE TEXT 7, 0
```

VDP SPRITES ON / VDP SPRITES OFF

Enables or disables hardware sprites on the current base mode by re-sending command 11 with or without the sprite flag (`4`). The transpiler keeps the last selected base mode (`BITMAP`, `TILE`, or `TEXT`) and, for text mode, the last foreground/background colors.

```
10 VDP MODE BITMAP
20 VDP SPRITES ON
30 VDP SPRITES OFF
```

VDP CLS

Fills the 16x8 text grid with spaces. Equivalent to command 14 with character code 0. Use after `VDP MODE TEXT`.

```
10 VDP CLS
```

VDP TEXTFILL

Fills the 16x8 text grid with one ROM character code. Equivalent to command 14 with the selected code. `VDP CLS` is the same as `VDP TEXTFILL 0`.

```
10 VDP TEXTFILL 37
```

VDP TEXT

Writes one text cell at `col`, `row` with a compile-time ROM character code `0..63`. Equivalent to command 12 with the four DATA nibbles for column, row, code high, and code low.

```
10 VDP TEXT 0, 0, 18
```

VDP PRINT

Writes a string literal left to right on one text row. The transpiler expands each supported character to command 12 at compile time. The column must be a numeric literal when the string contains more than one character.

```
10 VDP PRINT 0, 0, "SCORE 07"
```

Supported characters match the VDP ROM font: space, digits, `A..Z`, and `. , : ; ! ? - + = / ( )`.

VDP CLEAR

Clears the back buffer to a palette index. Equivalent to `POKE @ROMPORT(8), 4` followed by one DATA color nibble.

```
10 VDP CLEAR 12
```

VDP PLOT

Plots one pixel at `x`, `y` with a palette color. Equivalent to command 5 with X/Y high and low nibbles.

```
10 VDP PLOT 32, 24, 8
```

VDP LINE

Draws a line segment. Equivalent to command 6.

```
10 VDP LINE 4, 4, 60, 44, 12
```

VDP RECT

Draws a hollow rectangle. Equivalent to command 7.

```
10 VDP RECT 8, 8, 48, 32, 9
```

VDP FILLRECT

Draws a filled rectangle. Equivalent to command 8.

```
10 VDP FILLRECT 20, 18, 24, 12, 8
```

VDP CIRCLE

Draws a hollow circle. Equivalent to command 9 in bitmap mode.

```
10 VDP CIRCLE 32, 24, 12, 12
```

VDP FILLCIRCLE

Draws a filled circle. Equivalent to command 10 in bitmap mode.

```
10 VDP FILLCIRCLE 32, 24, 8, 9
```

VDP TILE

Writes one tile-map cell in tile mode. Equivalent to command 12 with column, row, and pattern index.

```
10 VDP TILE 0, 0, 1
```

VDP FILLMAP

Fills all 128 map cells with the same pattern index. Equivalent to command 14.

```
10 VDP FILLMAP 0
```

VDP SCROLL / VDP SCROLLHOME

Moves the tile viewport in pixels or resets it to the origin. Equivalent to commands 2 and 3 in tile mode.

```
10 VDP SCROLL 8, 0
20 VDP SCROLLHOME
```

VDP PATTERN / VDP PATTERNHOME / VDP PATTERNADDR / VDP PATTERNDATA

Pattern-generator helpers for tile mode. `VDP PATTERN id` opens command 13 for one pattern index. `VDP PATTERNHOME` homes the stream at pixel 0. `VDP PATTERNADDR index` selects a 12-bit stream address with command 9. `VDP PATTERNDATA color` writes one autoincrementing pattern pixel nibble on DATA.

```
10 VDP PATTERNHOME
20 VDP PATTERNDATA 8
30 VDP PATTERN 1
```

VDP SPRITE

Updates one hardware sprite in bitmap mode with sprites enabled. Equivalent to command 12 with eight DATA nibbles: sprite index, Y high, Y low, X high, X low, pattern high, pattern low, and color.

```
10 VDP MODE BITMAP
20 VDP SPRITES ON
30 VDP SPRITE 0, 20, 16, 1, 8
```

VDP SPRITEHIDE

Hides one sprite by sending `Y = 255` in the sprite update protocol. Equivalent to command 12 with the selected sprite index and hidden Y.

```
10 VDP SPRITEHIDE 0
```

VDP STATUS

Reads the VDP STATUS port into a variable. Equivalent to `PEEK @ROMPORT(9), variable`. Bit 0 is VBLANK; bit 1 reports more than four sprites on one scanline; bit 2 reports sprite overlap. Reading STATUS clears the latched bits until the next event.

```
10 DIM V AS NIBBLE
20 VDP STATUS V
```

VDP TEST

Loads the built-in VDP test pattern. Equivalent to `POKE @ROMPORT(8), 1`.

```
10 VDP TEST
```

`VDP PLOT`, `VDP SPRITE`, `VDP SCROLL`, and `VDP TILE` accept runtime variables for coordinates, scroll offsets, and pattern indexes. Coordinates and scroll offsets are emitted as two DATA nibbles from a scalar expression (typically `BYTE`). Pattern indexes use the same two-nibble encoding as tile character codes. Other VDP drawing commands still require compile-time literals for coordinates, sizes, radii, and stream addresses in the current transpiler.

Examples

Example 1: QuadBasic Program

```
10 DIM A AS NIBBLE
20 DIM B AS NIBBLE
30 A = 5
40 B = 10
50 PRINT "A="; A
60 PRINT "B="; B
70 PRINT "Sum="; A + B
80 END
```

Example 2: FOR Loop

```
10 DIM I AS NIBBLE
20 FOR I = 1 TO 10
30   PRINT I
40 NEXT I
50 END
```

Example 3: IF-THEN

```
10 DIM X AS NIBBLE
20 X = RND()
30 IF X > 7 THEN
40   PRINT "Greater than 7"
50   GOTO 70
60 ENDIF
70 PRINT "End"
80 END
```

Example 4: Subroutine

```
10 PRINT "Start"
20 GOSUB 100
30 PRINT "Returned"
40 END

100 PRINT "In subroutine"
110 RETURN
```

Example 5: Memory Access

```
10 DIM A AS NIBBLE
20 DIM B AS NIBBLE
30 REM Write to RAM
40 POKE @RAM(0, 0, 0, 0), 5
50 POKE @RAM(0, 0, 0, 1), 10
60 REM Read from RAM
70 PEEK @RAM(0, 0, 0, 0), A
80 PEEK @RAM(0, 0, 0, 1), B
90 PRINT "A="; A
100 PRINT "B="; B
110 END
```

Example 6: Logical Operations

```
10 DIM A AS NIBBLE
20 DIM B AS NIBBLE
30 DIM RESULT AS NIBBLE
40 A = 5
50 B = 10
60 RESULT = A AND B
70 PRINT "AND:"; RESULT
80 RESULT = A OR B
90 PRINT "OR:"; RESULT
100 RESULT = A XOR B
110 PRINT "XOR:"; RESULT
120 RESULT = NOT A
130 PRINT "NOT A:"; RESULT
140 END
```

Example 7: PRINT with Multiple Devices

```
10 PRINT "Screen"
20 PRINT @PRN, "Printer"
30 PRINT @LCD, "LCD"
40 PRINT @NUL, "Discarded"
50 END
```

Example 8: Complex Conditions

```
10 DIM A AS NIBBLE
20 DIM B AS NIBBLE
30 DIM C AS NIBBLE
40 A = 5
50 B = 10
60 C = 3
70 IF A = 5 AND B > 5 THEN
80   PRINT "Condition 1"
90 ENDIF
100 IF A < 10 OR C = 3 THEN
110  PRINT "Condition 2"
120 ENDIF
130 IF NOT A = 0 THEN
140  PRINT "A is not zero"
150 ENDIF
160 END
```

Important Notes

1. **Line numbers:** All lines must start with a number (0-65535)
2. **Typed variables:** Every variable needs `DIM` and a type
3. **Case-insensitive:** The language does not distinguish between uppercase and lowercase
4. **Statement separators:** Use `:` to separate multiple statements on the same line
5. **Comments:** Can use `REM` or `'` for comments
6. **Nested IFs:** Supported in block bodies and as the single statement of an inline IF (including nested IF ... THEN on one line). Each block IF must close with its own ENDIF.
7. **Maximum comparisons:** IF conditions can have up to 8 comparisons
8. **CHR only in PRINT:** The `CHR()` function can only be used inside `PRINT`
9. **PEEK @RAMPORT:** Not supported (RAM port is write-only)
10. **Arithmetic `*` and `/`:** Valid in source for binary types; lowered to `ADD`/`SUB` loops in ASM
11. `@COM` / `@AUX`: Reserved for future extensions; not valid in `PRINT` until the language accepts them
12. **Every variable must be declared.** There are no implicit declarations.
13. **All integer types are unsigned.** The 4004 has no native signed arithmetic.
14. **Storage is deterministic.** The Variable Map shows where each variable lives.
15. `DIM` **is header-only.** Declarations must precede executable statements.
16. **Initialisation is explicit.** Assign after `DIM`; RAM types may get a boot zero-fill prologue.
17. **Casts are mandatory between distinct types.** No silent conversions in expressions.
18. **BCD and binary do not mix without an explicit cast.**
19. **System RAM on the last chip** (`RAM[3, 3, 2, *]` and `RAM[3, 3, 3, *]`) is reserved; see Reserved system RAM.
20. **Arrays are 1-dimensional and not bounds-checked.**
21. `DEC` / `HEX` / `CHR`: Only valid inside `PRINT` items, not as general expressions.
22. **Page-F runtime budget:** Linked helpers must fit in **256 bytes** of ROM page F.
23. **Binary → BCD casts** (`BCD2`…`BCD16`) use inline mainline code; **BCD → binary** (`BYTE(bcd)`, `NIBBLE(bcd)`) calls `B4_BCD2BIN` in page F when not constant-folded.

Quick Reference

Keywords by Category

Control: IF, THEN, ENDIF, END, GOTO, GOSUB, RETURN
Loops: FOR, TO, STEP, NEXT
Declarations: DIM, CONST, LET, REM
I/O: PRINT, CHR, DEC, HEX, CLS
Memory: PEEK, POKE
Logical Operators: NOT, AND, OR, XOR, NAND, NOR, XNOR
Functions: RND
Types: `NIBBLE`, `BYTE`, `BCD(n)`, arrays

Types

Type	Bits	Range	Default Storage	Working registers (during ops)
`NIBBLE`	4	0..15	R0..R9	Same register
`BYTE`	8	0..255	RAM 4002	R14 (low), R15 (high)
`BCD(n)` (n=1..16)	4n	0.. (10^n−1)	RAM 4002	
`T(N)` (array)	N × size(T)	–	RAM 4002	

Declaration

```
DIM Name AS Type
DIM Name(N) AS Type
DIM A, B, C AS Type
CONST Name [ AS Type ] = Literal
```

Operators by Type

See Arithmetic in Source and in ASM for the distinction between operators in source and lowering in ASM.

Type	Arithmetic (source)	Logical	Comparison
`NIBBLE`	`+ - * / -unary`	`AND OR XOR NAND NOR XNOR NOT`	all
`BYTE`	`+ - * / -unary`	`AND OR XOR NOT`	all
`BCD(n)`	`+ -`	(none)	all

For `NIBBLE` and `BYTE`, `*` and `/` in the **Arithmetic (source)** column are first-class operators in QuadBasic. The transpiler implements them with visible `ADD`/`SUB` loops in the listing, not with 4004 multiply or divide instructions.

Casts

Cast	Target
`NIBBLE(x)`	4 bits
`BYTE(x)`	8 bits
`BCD2(x)` .. `BCD16(x)`	n-digit BCD

Reserved RAM (off-limits to automatic allocation)

Address	Owner	Use
`RAM[3,3,2,0..15]`	Compiler	Work area on last chip
`RAM[3,3,3,0]`	Compiler	`BYTE` carry
`RAM[3,3,3,1..12]`	Compiler	Temps / `BCD` helpers / spill
`RAM[3,3,3,13]`	Runtime	`PRINT` device id
`RAM[3,3,3,14..15]`	Runtime	`RND` state

Operators by Precedence

1. `()` (parentheses)
2. `NOT`, `-` (unary)
3. `*`, `/`
4. `+`, `-`
5. `AND`, `OR`, `XOR`, `NAND`, `NOR`, `XNOR`

Appendix: QuadBasic Runtime Routines

QuadBasic links helper routines in **ROM page F** (`0F00H`–`0FFFH`). Page 0 holds **stubs** (`B4_AND:` … `JUN B4_AND_F`) so program code can `JMS` a short local name; the body lives in page F (`B4_AND_F:` …).

Only routines your program needs are emitted. The combined runtime must fit in **256 bytes** of page F or transpilation fails.

Runtime catalog (current transpiler)

Stub (page 0)	Body (page F)	Linked when
`B4_NOT`	`B4_NOT_F`	`NOT` in expressions
`B4_AND`	`B4_AND_F`	`AND`
`B4_OR`	`B4_OR_F`	`OR`
`B4_XOR`	`B4_XOR_F`	`XOR`, and internally by `B4_RND_F`
`B4_NAND`	`B4_NAND_F`	`NAND`
`B4_NOR`	`B4_NOR_F`	`NOR`
`B4_XNOR`	`B4_XNOR_F`	`XNOR`
`B4_RND`	`B4_RND_F`	`RND()`
`B4_PRINTDEC`	`B4_PRINTDEC_F`	`PRINT` / `PRINT DEC`, per-digit output, some `BCD` paths
`B4_PRINTCHAR`	`B4_PRINTCHAR_F`	strings, `CHR`, `CLS`, called from `B4_PRINTDEC_F` / `B4_PRINTHEXNYB_F`
`B4_PRINTHEXNYB`	`B4_PRINTHEXNYB_F`	`PRINT HEX(...)`
`B4_PRINTLF`	`B4_PRINTLF_F`	trailing newline after `PRINT`
`B4_SETDEV`	`B4_SETDEV_F`	`PRINT @device`, `CLS`
`B4_BCD2BIN`	`B4_BCD2BIN_F`	`NIBBLE(bcd)` / `BYTE(bcd)` casts

`B4_STROBE_F` exists in the runtime source but is **not linked** by the current transpiler: **STROBE** (ROM0 bit 0 low→high) is performed **inline** inside `B4_PRINTCHAR_F` and `B4_PRINTLF_F` to save hardware stack levels on the 4004.

Emission order in page F (when multiple routines are present): logical ops → `RND` → `STROBE` (if ever linked) → `PRINTDEC` → `PRINTCHAR` → `PRINTHEXNYB` → `PRINTLF` → `SETDEV` → `BCD2BIN`.

Logical operations (page F)

B4_NOT_F

- **Input**: ACC = nibble (0..15)
- **Output**: R10 = bitwise NOT; ACC = 0 on return (`BBL 0`)
- **Used by**: `NOT` in expressions (`JMS B4_NOT` from generated code)

B4_AND_F / B4_OR_F / B4_XOR_F

- **Input**: R10 = A, R11 = B
- **Output**: R10 = result nibble; ACC = 0 on return
- **Used by**: `AND`, `OR`, `XOR` in expressions

B4_NAND_F / B4_NOR_F / B4_XNOR_F

- **Input**: R10 = A, R11 = B
- **Output**: R10 = NAND / NOR / XNOR; ACC = 0 on return
- **Implementation**: `JMS B4_AND_F` / `B4_OR_F` / `B4_XOR_F`, then `CMA` on the result in ACC

Random number generation

B4_RND_F

- **Input**: ACC = argument (ignored today)
- **Output**: R10 = pseudo-random nibble 0..15
- **State**: `RAM[3, 3, 3, 14]` (low), `RAM[3, 3, 3, 15]` (high) — updated each call
- **Algorithm**: 8-bit state += 5 (mod 256), then `out = ((low+high) mod 16) XOR ((high<<1) mod 16)` via `JMS B4_XOR_F`
- **Used by**: `RND()` (`JMS B4_RND`; result read from R10)

PRINT runtime (page F)

Fixed wiring for character output:

- **ROM0.PORT** bit 0 = STROBE (0 then 1)
- **ROM1.PORT** = ASCII high nibble
- **ROM2.PORT** = ASCII low nibble

B4_PRINTCHAR_F

- **Input**: R11 = HI nibble, R10 = LO nibble
- **Action**: writes ROM2, ROM1, then **inlined STROBE** on ROM0 (no `JMS B4_STROBE`)
- **Used by**: each `PRINT` character, `PRINT CHR`, `CLS` control codes, and internally from `B4_PRINTDEC_F` / `B4_PRINTHEXNYB_F`

B4_PRINTLF_F

- **Input**: none
- **Action**: emits LF (`HI=0`, `LO=10`) with **inlined STROBE**
- **Used by**: `PRINT` without trailing `;`, `PRINT ""`

B4_PRINTDEC_F

- **Input**: ACC = value 0..15 (preserves R14/R15)
- **Action**: one ASCII digit `0`..`9`, or two digits `10`..`15`, via `JMS B4_PRINTCHAR`
- **Used by**: `PRINT DEC(expr)`, each decimal digit when printing wider values, and some `BCD` display helpers

For a full `BYTE` (0..255), the transpiler usually emits **mainline** division loops that call `B4_PRINTDEC` once per digit; **R14**/**R15** hold the working copy during that sequence. The stored value remains in RAM.

B4_PRINTHEXNYB_F

- **Input**: ACC = nibble 0..15
- **Action**: one hex ASCII character `0`..`9` or `A`..`F` via `JMS B4_PRINTCHAR`
- **Used by**: `PRINT HEX(expr)` (`NIBBLE`: one call; `BYTE`: two calls, **high nibble first**)

BCD to binary runtime

B4_BCD2BIN_F

- **Input** (set up by generated cast code before `JMS B4_BCD2BIN`):
  - `DCL` bank already selected
  - `R12:R13` = `SRC` address of the **least significant** BCD digit (via `FIM`/`SRC`)
  - `R7` = BCD digit count (1..4 for current casts)
- **Output**: binary accumulators in **R14** (low), **R15** (high), **R8** (extension for values past 255 during conversion); `BYTE` destinations store **R14:R15** to RAM
- **Used by**: `NIBBLE(bcd_var)` / `BYTE(bcd_var)` casts (MCS-4-style decimal-to-binary). **Not** used for `BCD2`…`BCD16` (binary→BCD is inline mainline code).

Device selection

B4_SETDEV_F

- **Input**: ACC = device id 0..6
  - 0 = SCR, 1 = PRN, 2 = LCD, 3 = DSK, 4 = COM (reserved), 5 = AUX (reserved), 6 = NUL
- **State**: `RAM[3, 3, 3, 13]` — active `PRINT` device (written only when id changes)
- **Used by**: `PRINT @device`, `CLS` (restores previous device after clear)

Runtime organization

- **Stubs** on page 0: `B4_xxx:` + `JUN B4_xxx_F`
- **Bodies** on page F: `ORG 0F00H` then linked routines only
- **Far calls**: `JMS` / `JUN` across pages (no `JCN` across page boundary)
- **Register contract**: routines preserve **R0..R9** (user `NIBBLE` variables). **R10..R15** are scratch. Do not keep program data in R10..R15 across a runtime call.
- **System RAM**: runtime uses `RAM[3, 3, 3, 13..15]`; compiler scratch uses the same last chip (registers 2 and 3). See System RAM on the last chip.

`B4_INIT` (RAM zero-fill for `BYTE`/`BCD`/arrays) is **compiler prologue** on page 0, not part of the page-F runtime library.

Important notes

1. **ROM 0–2** are dedicated to the print interface in generated programs; do not assign external peripherals to those chips in the default print layout.
2. **STROBE** still occurs on every character and newline, but it is **inlined** in `B4_PRINTCHAR_F` / `B4_PRINTLF_F`, not via a separate `JMS B4_STROBE`.
3. **Dependencies**: `B4_NAND_F` → `B4_AND_F`; `B4_NOR_F` → `B4_OR_F`; `B4_XNOR_F` → `B4_XOR_F`; `B4_RND_F` → `B4_XOR_F`; `B4_PRINTDEC_F` / `B4_PRINTHEXNYB_F` → `B4_PRINTCHAR_F`.
4. Linking `B4_XOR_F` for `RND()` also pulls the `XOR` stub if `XOR` is not otherwise used in source.

LEGAL NOTICE

> **Legal Notice and Fair Use Disclaimer:**

This project, Quadium 4004 Workbench, is an independent educational tool developed for historical preservation, research, and instructional purposes regarding early computing architectures.

Trademark Acknowledgement: Intel, MCS-4, 4001, 4002, 4003, and 4004 are registered trademarks of Intel Corporation. Texas Instruments, TMS9918, and related VDP marks may be trademarks of Texas Instruments Incorporated. These terms are used herein solely for nominative purposes to describe technical compatibility and historical context, which constitutes "fair use" under intellectual property laws. All references to these products are made for descriptive and compatibility purposes only.

Non-Affiliation: This project is not affiliated with, authorized, sponsored, or endorsed by Intel Corporation or Texas Instruments Incorporated.

Original Documentation: This documentation is an independent work created for Quadium 4004 Workbench. QuadBasic is an original language implementation designed for educational use with the emulated MCS-4 system.

Third-party marks and materials: Intel, MCS-4, 4004, 4001, 4002, and 4003 may be trademarks of Intel Corporation. Texas Instruments, TMS9918, and related VDP marks may be trademarks of Texas Instruments Incorporated. This product is not affiliated with or endorsed by Intel or Texas Instruments. Quadium does not distribute third-party ROM dumps, Intel manuals, Texas Instruments manuals, or datasheets. QuadBasic sample programs supplied with the product are authored by the publisher and transpile to 4004 assembly. Users are responsible for the rights to any files they load into the simulator.

Nature of Simulation: All logic described in this documentation represents an independent software implementation of publicly documented hardware specifications.

Document version: 0.80 (WIP)
Date: May 2026
Language: QuadBasic