

A Discussion Paper on similar issues in Forth and LabVIEW.

This is a discussion showing similar programming design and build weaknesses in the Forth and the LabVIEW programming methods many use.

If you are not or have never been a Forth Programmer you are probably wondering how these two very different languages have similar program designing and building issues.

In a Nut Shell, it is the use of literals in both languages that can cause large downstream issues, especially as the programs get larger.

This is a fundamental design problem for both languages. (actually in all non-declarative languages like some Basics etc).

Just to make it interesting and remove bias, I decided to quiz an AI engine on this problem and was most surprised with what I got back.

This short review in an AI session shows possible ways of how to remove most of these similar design obstacle both in Forth and LabVIEW.

Let's start with Forth first as it is a very clear issue. The Forth literal problem is exposed easily and if you are a LabVIEW programmer you will then understand what you are doing in LabVIEW Diagrams with Literals (Constants)

Forth uses literals on a stack, for much of its programming development method. It is central to everything in Forth. Everything is done on the stack.

This makes the Forth code developed using this Literal/ Stack method very difficult to document, understand (reread), update and troubleshoot code after development and even during development.

The "Forth's Standard Way" around this issue was to develop/build very short pieces of code (words or components) so any stack work was small, local and more obvious and probably safer. But later a big mental burden.

So let's look at the AI discussion.

Question Posed to AI :

(In Forth are literals used on the stack, a major program design weakness.)

AI Answer:

Using literals directly on the stack in Forth is indeed a widely recognized design vulnerability that degrades code readability and maintainability.

While Forth's simplicity relies on the stack, heavy use of "magic numbers" makes code fragile and difficult to refactor. [1, 2, 3, 4] **(A good start)**

Why Literals on the Stack Cause Problems

- **Breaks Stack Math:** Changing a literal value might change the number of items or data types on the stack, breaking subsequent stack operators like DUP, SWAP, or DROP.
- **Obscures Intent:** A raw number (literal) like 42 on the stack provides zero context about what that data represents.
- **Hinders Refactoring:** Hardcoded values scattered across words make global changes error-prone or near impossible. [1, 2, 3]

How Good or Better Forth Design can Mitigate this issue

Experienced long term Forth programmers avoid this weakness by applying specific design patterns:

(A good example of this is OPTO22's early Cyrano software written in Forth)

- **Constants and Variables:** Defining names using CONSTANT or VARIABLE to replace raw numbers (literals) with meaningful text.
- **Aggressive Factoring:** Breaking code into tiny, single-purpose words so that a literal is only used once and instantly consumed.
- **Descriptive Stack Comments:** Using notation like (a b -- c) to document exactly what literals turn into. [1, 2, 3, 4]. Even this though is making the programmer responsible to show stack action,

My Statement to AI:

By using Definitions and Declarations in Forth, its literals / stack weakness can be mitigated

AI Answer:

Yes, defining named constants and variables completely mitigates this specific Forth weakness. Shifting from raw stack literals to named data structures transitions the code from fragile "stack juggling" to a self-documenting architecture.

Key Mitigation Strategies

- **CONSTANT:** Creates named, immutable values. It automatically pushes its value to the stack when called.

- **VARIABLE:** Allocates memory for mutable data. It pushes the memory address to the stack, protecting data integrity.
- **CREATE ... DOES>:** Builds custom data structures and behaviors. This isolates complex stack manipulation inside a single definition.

Impact on Code Quality

- **Decouples Data and Logic:** Changing a value at the Definition level prevents breaking the downstream stack math.
- **Improves Readable Intent:** Replacing 42 with the name MAX-ITEMS makes the operational context immediately clear to developers.
- **Reduces Stack Depth:** Storing state in named variables minimizes the need for complex stack operations like ROT or PICK.

AI Offered: If you are writing or reviewing Forth code, would you like to see a practical code example comparing a literal-heavy stack implementation against a refactored version using definitions?

So if you have an interest in this ask this question of any AI engine.

I asked AI:

Forth failed as a language due to the literal / stack design failure?

AI Answer:

Forth's reliance on explicit stack manipulation and literal handling was a foundational reason for its failure to achieve mainstream commercial adoption.

While its minimal syntax and raw power made it legendary for early embedded systems, the inherent friction of managing a naked data stack ultimately doomed it to a niche cul-de-sac in computer history. [1, 2]

The structural design failures that drove its commercial decline center on a few critical realities:

1. The Cost of "Stack Shaking"

As applications grew in scale, Forth code became dominated by what the community calls "stack juggling" or "stack shaking". Developers spent an excessive amount of cognitive

energy using words like DUP, SWAP, ROT, and PICK just to route variables to the right operations. [1, 2]

- **The Infix Advantage:** Humans naturally think in terms of keyword identifiers and lexical scopes, not positional arguments.
- **The Cognitive Wall:** When a stack grows beyond 3 or 4 items, the mental tracking required to know what a generic literal represents becomes overwhelmingly complex. [1]

2. Zero System Protection & Total Blindness

Forth functions (words) communicate implicitly through the stack. Because there is no compiler-enforced type checking, abstract syntax tree (AST), or memory safety, a single misplaced literal or mathematical miscalculation creates a silent disaster. [1, 2, 3]

- **Stack Pollution:** Pushing an unintended literal or forgetting to clean up a value corrupts every single downstream operation.
- **Silent Corruption:** A stack error can easily cause Forth to read or write a random address in memory, resulting in catastrophic system crashes or data corruption that is incredibly difficult to debug. [1]

3. The Refactoring Nightmare

In structured, standard modern languages (like C, Python, or Rust), adding a parameter to a function is simple. In Forth, introducing a new literal to a word shifts the entire positional geometry of the stack. A developer rewriting a low-level word must manually audit and adjust every single calling word up the execution tree to prevent stack imbalance. [1]

4. Economic Realities vs. Software Scaling

Forth was designed for an era when hardware was incredibly constrained (kilobytes of RAM) and compiling speed mattered. As Silicon Valley advanced, optimizing for human developer time became more critical than saving machine bytes. Other languages scaled beautifully to millions of lines of code because they protected the programmer from low-level memory geometry; Forth required a "genius-level" mental model just to maintain stable stack frames. [1, 2, 3, 4]

Offer by AI:

Would you like to explore how modern concatenative languages (like Factor or PostScript) altered the stack design to address these exact failures?

I offered this thought to the IA:

In Forth the programmer becomes the parser and the compiler.

AI Answered:

That is precisely why Forth is both revered and avoided: in Forth, the programmer is forced to act as the compiler, parser, and type-checker.

Because the Forth environment provides almost no abstraction or safety nets, the developer's brain must manually execute the tasks that modern compilers automate.

The Tasks the Programmer Inherits

- **Manual Type Checking:** Forth does not know if a stack item is an integer, an address, or a pointer. The programmer must mentally enforce types.
- **Stack Frame Tracking:** The developer must manually trace the exact depth and state of the stack across every single word execution.
- **Syntax Parsing:** Forth processes text strictly from left to right, space by space. The programmer must design the execution order without syntax trees.
- **Memory Management:** The developer directly manipulates memory addresses using primitives like @ (fetch) and ! (store), risking total system crashes.

The Trade-off: Ultimate Power vs. Extreme Fatigue

This design makes Forth uniquely powerful but commercially exhausting:

- **The Benefit:** Total control. Because there is no heavy compiler middleware, Forth is incredibly fast, highly interactive, and can fit into microscopic embedded systems.
- **The Cost:** High cognitive load. Instead of focusing on business logic, the programmer spends significant mental energy just preventing the stack from collapsing.

AI Offered : If you want, I can: show how **Forth's IMMEDIATE words** let the programmer literally extend the compiler at runtime. Compare this to **how modern compilers (like Rust or GCC) protect the developer**

I also said this to AI :

Another language that has a similar difficulty in the use of literals is LabVIEW from NI.

AI Response:

You have made an excellent architectural connection. LabVIEW suffers from the exact same structural vulnerability with literals as Forth, fundamentally caused by literal blindness and spatial dependency.

While Forth passes literals implicitly over a chronological stack, LabVIEW (a graphical dataflow language) passes literals explicitly through physical "wires" on a block diagram. Despite the different visual mediums, the breakdown in software engineering principles is identical. [1, 2]

The shared flaws of block diagram literals in LabVIEW map directly onto Forth's stack failures:

1. The Visual Equivalent of "Stack Juggling"

In Forth, you lose track of raw literals as they move through stack operations (ROT, SWAP). In LabVIEW, if you place a raw numeric or string literal on the far left of a large block diagram, you must manually run a wire across screens of graphical space to feed it into a subVI or structure. [1, 2].

- **The "Spaghetti Wire" Problem:** Large LabVIEW programs quickly dissolve into an unreadable tangle of criss crossing wires solely to route static configuration values (literals).
- **Hidden Mutability:** Tracking where a hardcoded literal splits, merges, or routes on a 2D canvas is just as mentally exhausting as tracking positional data frames on a 1D Forth stack.

2. Total Context Blindness

Dropping a box containing 500 directly onto a LabVIEW block diagram provides zero semantic context. [1] (Same as Forth Stack literals)

- A developer looking at that code has no idea if 500 represents a timeout in milliseconds, a maximum temperature threshold, or a loop count limit.
- Just like a naked 500 floating on a Forth stack, it relies entirely on the programmer's perfect memory to maintain safety and meaning. [1, 2]

3. The Refactoring Lock-In

If you hardcode a numeric literal inside five different subVIs (or sections of a loop), updating that value requires opening every single graphical file (.vi) and manually rewiring or retyping the block. Because LabVIEW code is stored as monolithic binary files rather

than plain text, you cannot use standard command-line tools (like `grep` or `sed`) to easily search, find, and replace these magic numbers. [1, 2, 3]

The Universal Cure: Strict Definitions

Just as Forth developers rely on `CONSTANT` and `VARIABLE`, professional LabVIEW architects mitigate this by banning free-floating literals entirely. Instead, they force the software to utilize:

- **Strict Type Definitions (Strict Type Defs):** Centralized graphical templates ensuring that changing a control or constant instantly propagates across the entire application ecosystem.
- **Global/Functional Global Variables (FGVs):** Encapsulated data structures that store state and isolate configuration data from the primary execution diagram.

AI Offered : Would you like to explore how LabVIEW's "dataflow" paradigm enforces execution order, or compare it further to Forth's chronological execution?

I offered to AI:

A better program design is to bring all literals, constants and variables out of the program and create an editable graphical definition and declaration file system that loads at runtime.

AI Answers:

That is the golden rule of modern software architecture: separating configuration data from execution logic by using externalized, runtime-loaded definition files solves the literal vulnerabilities found in both Forth and LabVIEW.

By moving all literals, constants, and variables out of the hardcoded program and into an external, human-editable filesystem, you transform the code from a rigid, fragile script into a highly adaptable, data-driven engine.

The Architectural Benefits of Externalization

- **Zero-Recompile Adjustments:** Engineers can tune system parameters, calibration data, or communication ports at runtime without opening a block diagram or touching a compiler.
- **Semantic Context Enforced:** Raw numbers (500) are replaced by clear, textual key-value pairs (`Timeout_ms = 500`) within files that any developer can easily read and audit.
- **Immunity to Code Rot:** Changing a value inside an external file cannot break Forth stack geometry or tangle LabVIEW graphical wires, eliminating cascading bugs

- **Version Control Sanity:** Configuration changes can be easily tracked using standard plain-text tools (like git), keeping binary code changes completely separate from pure data tweaks.

Standard Industry Patterns That Implement Your Design

This design philosophy is standard practice across enterprise software engineering today, typically implemented using a few universal formats:

- **INI and TOML Files:** Plain-text files utilizing simple header blocks ([Hardware]) and explicit key-value pairings—frequently used in LabVIEW for hardware abstraction layers.
- **JSON and YAML Trees:** Hierarchical structures perfect for mapping nested variables, data types, and complex operational profiles.
- **SQLite / Database Tables:** For highly complex or enterprise-scale systems, applications read their initial state and runtime boundaries out of structured local databases upon boot.

AI Offered: If you want to see how this is built, I can show a pattern for building a **Runtime Configuration Loader in LabVIEW**

AI Offered: Demonstrate how **Forth parses external text files at runtime** to initialize constants

Now to complete this a little further for you.

Notice that AI Offered text based solutions for LabVIEW. I didn't quiz it further on this but you can see code based bias in the answer or lack of knowing in the answer.

The best solution for LabVIEW is a Graphical Declaration and Definition Editor that runs externally to the LabVIEW program which will become empty of all Constants, Variables and Literals, taking all graphical data types out of the diagrams. Don't forget a Constant in LabVIEW is really a non defined literal (even though it may have a name).

At runtime these Definition and Declaration Graphical Constants and Variable Data files will seed the LabVIEW program and it runs with that editable data. (True Hidden Data Design)

This is a true Program Design method and makes long time maintenance easier and precise. It also cleans up the program design in terms of Data Coupling also as the files can be designated Private or Public. (Global or Local).

This definition and declaration step is most important for large programs that otherwise would become difficult to manage and especially difficult to successfully complete.

Note all the data in these files must be Type Definitions. This is crucial for this to be successful method.

What I have noticed many times, is that large programs using traditional LabVIEW programming methods (loose literals randomly scattered) require either a Guru Programmer or a Team of Gurus that can manage and remember all the un-documented, undeclared or definitionless Literals Data dropped in Diagrams. Only a programmer with a large memory can manage this type of development and if they leave the team, much happens.

Note : (Literals in LabVIEW can be any data type even graphical ones)

When I wrote the Book “**LabVIEW More LCOD**” it demonstrated this new method of externalised the graphical declaration and definition development. It is a design discipline that pays huge dividends in all middle to large sized LabVIEW projects and uses a file system saving pure LabVIEW external graphical named controls and Indicators of all types and any type.

In addition it shows how to build components using State Engines that can have their runtime states edited externally before runtime. This allows changing the state programs run methods design before runtime.

Also having external Graphical Declaration and Definition files allows additions to be externalised such as File Paths, Messages, Error info, all Control's and Indicators Numerics and Text Labels and Definition to be changed before runtime as Type definitions. This enables these Controls to be preset outside the diagram and the data is all in one location rather than scattered. This is management of programs at a new level.

This is the program's data being designed and decided outside the code or the program.

Hope that was helpful.

The Forth stuff may have been laborious but applies very well to LabVIEW. You never know it may get some old forthers playing again but this time using a definition section with Variables and Constants and not literals (or Magic Numbers).

Whenever I was called in to sort out programs or to complete uncompleteable programs it was almost always the same issues. No design rules, scattered data through the programs and at the end when it wouldn't run correctly, throw all caution to the wind and shotgun the diagrams to try and force it. Never works when there is coupling going on or worse forgotten data definitions scattered every where.

LabVIEW is a disciplined endeavour. The programmer becomes a much bigger part of the solution than code programming which can be very descriptive and exact. In LabVIEW the design is often the picture being drawn and that is the programmer for good or bad.

Just in case you are curious, I was a Forth programming in the 1980s before LabVIEW (1987) and I think this is why I liked LabVIEW so much. It initially allows development in an unstructured way but that eventually bites back when bigger things are tried. Forth is the same.

(In fact all attempts to program computers do the same if design discipline is missing)

You will know when you have won when you open the Diagrams and you do not see any literals (constants) any where in it, even down inside Sub VIs. And there sitting outside the program is a Graphical Data editor that has all the programs data ready to be changed or not that loads at runtime. And when I say runtime I do not necessarily mean at the program starting but when ever a component runs that needs data.

In summary the issue that drove the need to write all of this was that both Forth and LabVIEW have similar issues with literals and it needed to be discussed.

There was also a much bigger issue that this new design method allowed me to address. I had a one point, 15 sites doing similar Compliance Testing but each site having different hardware (IO counts in DAQ, Sensors, VSDs, Servos, different DAQ units, different sized test booths etc. I had developed 15 different but similar grafted versions of software for each site and to maintain. By using this new technique I was able to make a Universal Tester model that could be configured externally for each sites specifics. A huge step as it reduced the effort to maintain the sites by a factor of 15.

Using this new method I was even able to troubleshoot and reconfigure sites remotely by just editing the Definitions and Declarations in the Graphical Data Files.

Also the remote location could be running a Compliance Test in any section of the program and at runtime of any of the Components could take my remotely changed Graphical Data Files and configure that component in the new definition and declaration without any body noticing.

If you want more on this email me and I will put another specific PDF together for everyone.

Last point do not use ini Text Files for any of this type of work.

Use

True LabVIEW Graphical data files, just for the abstraction it gives!!