Jasm technical overview

admin (at) Jasm Tech (dot) com

Jasm *(noun)*: Take thunder & lightning, a steamship & a buzz-saw; mix them together, and put them in a woman. “She has jasm”. The term originated in 1842. Synonyms: Spirit, energy, spunk, zest.

Introduction

I have made a career-long project of finding the best tools, structures, and practices for preventing bugs and making code easier to understand. Making a language designed for this simplifies things greatly.

We write computer programs by typing text into an editor and trying to imagine what it will do. It is hard to program this way, and it is easy to make errors. This next generation language describes a better way.

The need for this product is great. The largest and most complex structures ever built by man are all software. Anything that makes big software easier to write and understand will generate tremendous savings.

The details of the language are about implementing best practices automatically and making the programmers job easier. The language is a vehicle for doing that in a way that is easy to write, read, and understand.

What is Jasm

Jasm is a proposed general-purpose programming language based on these principles:

Simplicity: Simplicity means easy to learn, easy to read, easy to program, easy to debug, and less complex.

Integrity: High integrity languages prevent bugs by performing data checks and disallowing dangerous practices.

Multitasking: Jasm simplifies multitasking & distributed programming.

Graphical IDE: Visualization tools and a graphical IDE make the programmers job easier (the Jasm Studio). You never edit a text file.

Powerful: Jasm has powerful high-level objects and operations.

Portable: Maximum portability.

Productivity: The end goal is more code done better in less time.

It does not look like any existing programming language, but all of the parts are well known and understood. Jasm is easy to understand.
Jasm invents some new things out of necessity, but mostly it combines existing concepts in new ways to do things better. That makes it easier to adopt.
Programmers want a language like the ones they already use, but better, gets it done twice as fast, and makes their lives easier. That is Jasm in a nutshell.

Jasm will allow large project completion in half the time of a hypothetical average language.

Jasm falls into these categories

- General purpose workhorse
- Procedural
- Object oriented
- Compiled
- Memory safe
- Explicit
- Strongly typed
- Concurrent
- High-integrity
- Uses a bytecode engine or transpiles to C++ or C#
- Both imperative and functional
- Off-side (indentation, not curly-braces)
- Big endian / Big Indian / Network order

The key is simplicity

- Easy to read & understand
- Low learning effort
- Easy to use
- Easy to maintain
- Makes best practices natural (a key design goal)
- Reduces software complexity
- Jasm is a low complexity language
- Hides all pointers and memory management; implements these in the simplest possible way.
- Low frustration (You should not have to stop programming to figure out how to do something, what this line of code does, or why it is doing that)
- Creates the boiler-plate for you (20-50% of the code is boiler-plate in most programming languages)
- Native support for security, distributed processing, and massive data sets
- Collapse/expand views hide unimportant details
- Doesn't give the programmer any hand-grenades. Simple, safe, easy-to-use methods replace all dangerous practices.
- Your time is worth more than a few bytes
- It is OK to have only one way to do something, as long as it is a good way
- Attention to the little details that prevent bugs and make your job easier

Creating simplicity is not simple, it has been my life's work.

High-integrity saves you debugging

Programmers spend more than half their time debugging, anything that reduces that is a big time-saver.
Jasm uses strictness, checking, and other high-integrity techniques because human error always happens.
High-integrity languages ensure that data will always be within range.
More than 50% of common bugs will not happen with Jasm, and more than 80% of the serious bugs will also be prevented.
High-integrity languages reduce the cost of development by 50% because there are 70% fewer fixes and 90% fewer bugs seen by the customer.

<u>The most common bugs</u>

- Procedure arguments not checked (Jasm checks the arguments for you).
- Boiler-plate errors (Jasm handles the boiler-plate for you)
- Syntax errors not caught (Jasm catches them)
- Errors handed incorrectly (Jasm has good error handling & messages)
- Test cases not keeping up with code changes (Jasm shows test coverage and lack thereof graphically)
- Flow control errors (Jasm prevents a lot of these with errors & warnings)
- Bad math (Sorry, Jasm can't fix your math)

The most difficult types of bugs to solve

- Pointer problems (Jasm does not have pointers)
- Memory deallocation problems (Jasm handles all memory allocation and deallocation for you)
- Access to memory that is no longer valid (Jasm prevents this)
- Buffer overruns (Jasm prevents this)
- Wrong order-of-initialization (Jasm separates construction from initialization and helps prevent order-of-initialization problems)
- Race conditions (Provides multitasking tools to help with this)

These can cause project-killing quagmires.

Other bug prevention

- Language design that makes best practices natural & automatic
- Language design produces simpler code
- High level objects and the ability to use validated 3rd-party libraries result in less code
- A lot of time has gone into the little details that prevent bugs and make the programmers life easier.

Designed for debug

Programmers spend half their time debugging; yet no language has been designed for debug before.
Most languages can debug, but you have to write code in special ways to do it effectively. Jasm does this for you.

Multi-tasking done right

The four types of multi-tasking are single-threaded, multi-threaded, multi-process, and distributed. Each is different from the other.
Jasm provides a seamless multitasking interface to all four, so the same code can run single-threaded, or use any type of multi-tasking in a blocking or non-blocking way.
Remote interfaces (TCP/IP, VXI, etc) of every type are virtualized, so your code handles all of them without any changes.
All of this is done in a simple and straightforward way.

Graphical IDE

A graphical Integrated Development Environment is something new, only one other language has it.

- Handles the usual IDE operations like editing, compiling, LINTing, debugging, threading, reformatting, boiler-plating, help, etc.
- Collapses details you don't care about right now, and expands them when you want them.
- You never have to edit text files; the IDE creates and hides all the boiler-plate (unless you want to look at it or debug through it).
- Some things are simpler and look better if viewed in a graphical way (see code examples below).
- The IDE handles transforming your code into modules, units (opaque modules with an interface), pre-compiled libraries, applications, projects, and so forth.
- Visualization tools are built into the IDE to help make sense of complex projects.
- The IDE hyperlinks everything for navigation and information.
- Handles important details like help, cleanup, metrics, spell-checking, language localization, over-compression/decompression, etc.
- The files are still easily readable in a 3rd party text editor.

Powerful

Provides high-level productivity-increasing commands and objects.
Half of the programming languages do not implement enumerations, and those that do implement them incompletely. Enumerations must be partially or completely hand-coded in all other languages.
No programming language implements symbol lists, state machines, binary search trees, language localization, or relational file managers. They must be hand-coded.
Jasm implements all of these things, and allows them to be extended to reduce program complexity.

Portable

The compiler & tools will be written in C++ for performance and portability. With two easy ports it will run on Windows, Linux/Unix, and Mac. This covers most compile platforms in use today.
Output can be set to one of several standard byte-code engines. This will allow execution just about everywhere.

Productivity

- Cuts development time in half by reducing bug rates and creating boiler-plate code automatically.
- Eliminates 80% of the worst types of bugs and prevents project-killing bug quagmires.
- This is the first language designed for debugging. Programmers spend more than half their time debugging.
- Programs are easier to understand.

Languages with similarities

- **Eiffel** is the most similar language. It is a high-integrity multi-tasking language with a graphical IDE. Eiffel is not used much because it is expensive ($10,000/seat), complex, and the programmer must learn to think in the Eiffel way. The free version of Eiffel is crippleware. *Jasm is better because it is much simpler, easier to learn, and we plan that it will be less expensive.*
- **Ada** is a high-integrity multi-tasking language. It is not used much because the core language, though well-designed, is old and archaic. It has been kept up to date, but updates were design-by-committee. It is also complex. Ada is a niche language used in defense work. *Jasm is better because it is simpler, more powerful, and better integrated.*
- **HAL/S** is an archaic high-integrity language. It was quite readable compared to other languages of its day. HAL/S was used in aerospace. *Jasm is better because it is a modern multitasking language.*
- **C#** has safety and multitasking built in, but at the cost of great complexity. When looking for a better programming language I was attracted to what C# can do until I saw the 527-page spec and said “I don’t want to learn this”. *Jasm is better because it is much simpler and easier to learn. Also, C# only gives safety to those who know what to do and not do.*
- **Go** has simplicity and multitasking, but does not significantly reduce bug rates. *Jasm is better because it reduces bug rates.*
- **Java** is highly portable and prevents the worst bugs, but does not prevent most bugs. It also needs a lot of boilerplate. *Jasm is better because it prevents common bugs and handles all the boilerplate.*
- **Rust** prevents the worst bugs, but does not prevent most bugs. It is also hard to learn. *Jasm is better because it is easier to learn and prevents common bugs.*
- **Kotlin** prevents the worst bugs, but does not prevent most bugs. *Jasm is better because it prevents common bugs.*
- **V** (aka **VLANG**) is comparable to the Jasm minimum viable product. *Jasm is better because it goes far beyond what VLANG does.*

No other language has the combination of simplicity and bug prevention that Jasm does. These things reduce development time and cost significantly.

What about functional programming?

Jasm has a functional programming mode. Use it whenever possible. You can cut back and forth between functional and imperative programming.
Jasm makes functional code easy to write & understand, and allows it to be used in most types of programming; This alone is worth making a new language.

What about purely functional languages?
FP uses a different programming paradigm. FP is simple, good at multitasking, and prevents most bugs. It's great if you can use it. Unfortunately, pure FP has these limitations:

- Pure FP programs are very difficult to design, read, and understand for those who are not practiced in the art. The learning curve is steep.
- It is not good for I/O, so not a good choice for programs that do a lot of it.
- It needs a lot of memory and CPU speed, so is not a good choice for systems that are limited in these areas.
- It doesn't do state well, so is not a good choice for programs that are fundamentally state machines.

I have been programming for over 40 years and did many types of programming. Every project I have been on did lots of I/O, had to fit into available memory, and was fundamentally a state machine. Functional programming has never been an option. This is largely why only one functional language has caught on, and only in its niche.
The best way to make money in pure FP seems to be to teach it; the ratio of jobs to practitioners is not good.

What about reactive programming?

Jasm has native support for reactive programming.

What about AI?

Jasm will benefit from artificial intelligence more than any other language.

- AI and no-code solutions promise to let you build applications much faster. In reality, they make easy things easier and hard things harder. Jasm makes hard things easier.
- Jasm gets the same benefit from AI as any other language.
- Jasm's syntax and structure allows AI-enabled programming better than most other languages.
- It will be easier to read & understand AI-generated code.
- We can support AI-enabled debugging, which does not have a lot of attention at this time.
- Jasm Studio is an integrated software development suite. We have a list of ways AI can assist and accelerate software development by cross-

connecting separate Jasm tools at a very low level. No other language can do this, and it allows Jasm programmers greater AI acceleration.

Pick one thing

If we had to pick only one thing to do well, it should be making a large multitasking/distributed application. Whatever the future of programming brings, it will probably be heavily parallel and/or distributed. Jasm pays a lot of attention to this target. Web development is the biggest chunk of the market right now.

Syntax summary

- Standard imperative programming language syntax. Most programmers will immediately recognize it, read it, and be able to come up to speed using it quickly.
- Standard object-oriented programming syntax.
- Attributes are declaration suffixes that modify the behavior of language elements. For example, the `noret` attribute tells Jasm that using the return value from your procedure is optional, and not to remind you if you don't use the return value from that procedure.
- Directives provide compiler instructions. For example, `#charsize` allows you to declare ASCII or UNICODE text.
- Pragmas are special comments that control error message output and Jasm tools. For example, `align-comma` pragmas line up the commas of your data into neat columns to make it easier to read.

Example definition file (**`temperature.jd`**)

- A definition file is a true header file.
- Comments are blue.
- **Keywords are bold.**
- Many things are simplified, like creating an enumeration by using the "create enum" tool; it creates a powerful object that can be derived from and extended.
- Because the procedure in this example takes an enum argument, it can handle data in 4 formats: `"Celsius", TemperatureUnits.CELSIUS, 0`, or `"0"`. Everything gets handled for you.
- `e` means it returns a standard error.
- The red argument is a reference that returns a value

```
// Water temperature may only be 0 to 100C
typedef float as water_temperature_in_c( 0.0 to 100.0 )
```

```
// Temperature can be in these units
Class TemperatureUnits is Enum:
    Index   Symbol       Name
    0       CELSIUS      "C"
    1       FAHRENHEIT   "F"
```

```
Convert water temperature into Celsius.
The answer is returned in temperatureInC.
Returns standard error code.
```

```
proc <err_ret> ConvertWaterTemperatureToC(
    temperatureInC is ref to water_temperature_in_c,
    value is float,
    units is TemperatureUnits.sym_type) ; e

#test

// Element test
proc <err_ret> Temperature_ElementTest( args is list of const text ) ; e
```

Example code file (**`temperature.jc`**)

- All the green text is supplied and kept in sync by Jasm
- Comments are blue.
- **Keywords are bold.**
- Jasm checks all the values and indexes for you
- The red argument is a reference that returns a value

```
proc <err_ret> ConvertWaterTemperatureToC(
    temperatureInC is ref to water_temperature_in_c,
    value is float,
    units is TemperatureUnits.sym_type):
     // Hidden checking for all arguments autogenerated here

    if (units == TemperatureUnits.CELSIUS):
          temperatureInC := value
    else: // (units == TemperatureUnits.FAHRENHEIT):
          temperatureInC := (value-32) * 5.0 / 9.0

     // Hidden checking for temperatureInC autogenerated here
    return ERROR_NONE


#test

// Element test
// Autogenerated element test code goes here
// Not included in this example
```