

> hi, I am now thinking about more precise features of monitor... I want to have switched protocol modes between interactive and host debugger and have UI adaptable at runtime to different widths of screen, expecting also 32cols as minimum, often 40cols, usually 80cols or even more, that's easy but I am more stuck by my thinking how to support in monitor the more cores, if it to be handled by the monitor or by some system calls as in multitasking system - you know, I want to be able to use more cores in VMEX/ASLIX apps but it must be easy and innovative, not complicated and manageable... definitely easier than any RTOS features, VMEX is for controlling things at quite high level, and cores/tasks must be handled intelligently and easily, predictably, may be even statically at quite high level of application, more to support things like async/await concept - simply to simplify code, not to be broken by handling too much different things in one code path, but to use just separate cores and their separate registers and allocated RAM area to allow cleaner code handling each separate task more easily than in single long spaghetti things just loudly thinking, don't be crazy, I am))))



It sounds like you're looking for a way to manage multiple cores and tasks in a simple, high-level manner for your VMEX/ASLIX applications. I understand that you want to avoid the complexity of a traditional RTOS while making it easy to switch between different modes and adapt the UI to various screen sizes.

Here's a potential solution for you to consider:

1. Create a high-level abstraction layer for managing cores and tasks:

Introduce an abstraction layer that allows you to create and manage tasks across multiple cores without worrying about the low-level details. This layer would handle task creation, scheduling, and synchronization, simplifying your application code.

2. Use an event-driven model for task management:

Instead of relying on traditional threading and synchronization mechanisms, use an event-driven model where tasks are triggered by events (e.g., a button press, sensor input, or timer). This can help you create cleaner, more modular code that is easier to manage and understand.

3. Support async/await-like constructs for easy concurrency:

Implement support for async/await-like constructs in your abstraction layer, allowing you to write asynchronous code more easily. This can help you to express complex

interactions between tasks in a more straightforward way, reducing the need for spaghetti code.

4. Use static allocation and task assignment:

Preallocate tasks to specific cores and allocate memory for each task at compile time. This can help you achieve deterministic performance, simplifying debugging and improving system stability.

5. Create a responsive UI that adapts to different screen sizes:

Design your monitor's UI to be adaptable at runtime to different screen widths. This will make it easier for users to work with your system, regardless of their display size.

6. Leverage system calls for multitasking when appropriate:

While your monitor should provide a high-level abstraction for managing tasks, it's still beneficial to have system calls for multitasking in place. This can help you to achieve better performance, especially when dealing with complex or resource-intensive tasks.

By following these guidelines, you should be able to create a simple, high-level system for managing multiple cores and tasks in your VMEX/ASLIX applications, avoiding the complexity of traditional RTOS solutions.



well, I understand ... and here I have my yesterdays brainstorming around it

? ... main help

? cmd ... cmd help

sp ... protocol mode interactive / compact

sr ... reg columns 4/2

smc ... mem columns 4/2

smr ... mem rows 1-20 hex

. ... lists all labels (a-z basic, alphanum extended)

.a ... get hex label a (a-z)

.a nnnnn ... set hex label a

pc ... get hex pc

pc nnnnnn ... set hex pc

r ... lists all regs

rn ...get hex reg n (1-F)
rsp, rra, rgp, rtp, rt0, rt1, rt2, rs0, rs1, ra0, ra1...
rn nnnnnn ... set hex reg n
rsp nnnnn ... set hex reg sp

m (lbl) nnn ... lists hex mem, by words (+/-)
m (lbl) aaaaaa nnn ... lists hex mem from aaaaaa
w (lbl) wwwwwwww.... hhhh.... bb.... set hex mem

g (lbl) aaaaaa ... go/continue
f (lbl) aaaaaa ... fn call with return (args a0,a1,...)
t (lbl) ... trace (by setting)
ti ... trace into
to ... trace over (fn calls)
tr ... trace return (from fn calls)

Memory will have defined areas by upper 16/8 bits the segments - flash, ram, eeprom, io (gpio, uart, pwm, spi, ...)... lengths will be MCU-specific, but general placements will be common by vmex

c ... list cores
cd nnn ... core debug (monitor switch)
cc ... core create
cs ... core save
cl ... core load
cr ... core remove
cp ... core priority
ct ... core trace



It seems like you've put a lot of thought into the commands and functionality of your monitor. Here's a summary of the commands you've brainstormed, along with a brief description of each command:

1. **?** - Display the main help menu
2. **? cmd** - Display the help menu for a specific command
3. **sp** - Switch protocol mode between interactive and compact
4. **sr** - Toggle register columns between 4 and 2

5. **smc** - Toggle memory columns between 4 and 2
6. **smr** - Set memory rows for hex display (1-20)
7. **.** - List all labels (basic and extended)
8. **.a** - Get hex label 'a'
9. **.a nnnnn** - Set hex label 'a' with value 'nnnnn'
10. **pc** - Get program counter (PC) value
11. **pc nnnnnn** - Set program counter (PC) value
12. **r** - List all registers
13. **rn** - Get hex value of register 'n'
14. **rn nnnnnn** - Set hex value of register 'n'
15. **m (1b1) nnn** - List memory by words with offset 'nnn'
16. **m (1b1) aaaaaa nnn** - List memory from address 'aaaaaa' with offset 'nnn'
17. **w (1b1) wwwwww.... hhhh.... bb....** - Set memory with hex values
18. **g (1b1) aaaaaa** - Go/continue execution from address 'aaaaaa'
19. **f (1b1) aaaaaa** - Function call with return, with arguments in registers a0, a1, etc.
20. **t (1b1)** - Trace execution with specified settings
21. **ti** - Trace into function calls
22. **to** - Trace over function calls
23. **tr** - Trace return from function calls
24. **c** - List cores
25. **cd nnn** - Core debug (monitor switch)
26. **cc** - Core create
27. **cs** - Core save
28. **c1** - Core load
29. **cr** - Core remove
30. **cp** - Core priority
31. **ct** - Core trace

Your brainstorming appears to cover a wide range of functionality for your monitor, including managing cores, tracing execution, and interacting with memory and registers. This command set should provide users with an extensive set of tools for debugging and managing their VMEX/ASLIX applications.





you know, I understand that I am quite naive also, or see some things too easily, and maybe, I am close to the parallax propeller (2) concept of cogs, to some extent, but I want to have possible dynamic creation of cores/tasks as bound to the application and available RAM of specific MCU, allow on simple ones simple things and on more powerful more things by MANY cores, just to save application from code/power to switch contexts other way than by switching cores by cooperative multitasking ... I think I need to think more, need more widespread experience what to allow in VEMX/ASLIX (and also how to introduce some asy syntax of AASLIX to handle this all, too)



It's completely normal to feel like you need to think more and gather more experience before implementing your ideas. Your concept of dynamically creating cores/tasks based on the available RAM and application needs is an interesting one. Here are a few thoughts to help guide your decision-making process:

1. Research existing solutions: Take a closer look at existing multi-core architectures and solutions like the Parallax Propeller (especially Propeller 2), as well as other cooperative multitasking systems. This research will help you understand the advantages and drawbacks of different approaches, and give you a broader perspective on how to design your own system.
2. Dynamic core creation: When implementing dynamic core creation, consider using a pool of pre-allocated cores/tasks that can be assigned to different applications as needed. This can help you avoid the overhead of creating and destroying cores/tasks at runtime, while still providing the flexibility to utilize the available resources efficiently.
3. Cooperative multitasking: Implementing cooperative multitasking in your system can help you achieve a balance between simplicity and performance. By allowing tasks to voluntarily yield control to other tasks, you can create a more predictable and manageable system without the need for complex scheduling algorithms.
4. Asynchronous syntax in AASLIX: Investigate how to introduce asynchronous syntax into AASLIX that simplifies the handling of concurrency in your applications. Some examples include `async/await`, lightweight threading, or other high-level constructs that abstract away the complexity of multi-core programming.
5. Scalability: Design your system to be easily scalable, allowing it to run on simpler, resource-constrained MCUs as well as more powerful ones with many cores. This will make it more versatile and attractive to a wider range of users.

6. Documentation and examples: As you design your system, make sure to document your ideas, architecture, and APIs thoroughly. Providing clear examples and use cases will help others understand how to utilize your system effectively.

By considering these aspects and continuing to refine your ideas, you'll be better equipped to design and implement a multi-core system for VMEX/ASLIX that meets your goals of simplicity, flexibility, and performance.



yes, many things are still open and I am aware of luxury to design things from scratch some practical way, ASLIX is only slightly higher-level portable assembler (in fact something like C but even more low level, build around RISC-V ISA/assembly as IR, expecting to transpile ASLIX to native assemblers of MCUs to allow to write low-level drivers etc... to use single language, simply as C ... but simpler, and more predictable, not having complexity and invisible differences of compiler internals



It's great that you have a clear vision for ASLIX as a portable, low-level language that is simpler and more predictable than C. Designing it around RISC-V ISA/assembly as an intermediate representation (IR) is a good choice, as it allows for easy translation to native assemblers for various MCUs.

Here are some suggestions to help you design ASLIX effectively:

1. Keep the language syntax and constructs simple: Focus on a small set of core language features and constructs that are easy to understand and use. This will make ASLIX more accessible to developers and easier to maintain and optimize.
2. Provide a clear mapping between ASLIX and RISC-V ISA: Make it easy for developers to understand how ASLIX maps to RISC-V ISA and assembly code. This will help them write efficient code and better understand the performance implications of their decisions.
3. Design for portability: Ensure that ASLIX is designed with portability in mind, so it can be easily adapted to different MCUs and architectures. This will make it more versatile and increase its potential user base.
4. Maintainability and predictability: Aim for a clean, modular design that is easy to maintain and understand. Predictable behavior is key to building a low-level language

that developers can trust.

5. Optimize for common use cases: Consider the most common use cases for your language, such as writing low-level drivers, and optimize ASLIX for these scenarios. This will make it more attractive to your target audience and increase its adoption.
6. Provide comprehensive documentation and examples: As with any programming language, clear documentation and example code are essential for helping developers learn and adopt ASLIX. Make sure to provide thorough documentation and examples covering all aspects of the language.

By following these guidelines, you can create a powerful, low-level language that meets your goals of simplicity, portability, and predictability, while also being well-suited for the development of low-level drivers and other embedded systems applications.



ya, ASLIX is almost 1:1 mapped to RISC-V assembly using C-operators, mostly; while natural assembly using instruction mnemonics is NOT readable, mapping to use C-operators and introducing light high level structures as loop(if)... again(if) if ... else ...endif (no for loops intentionally) provides "almost C look&feel" ...



It's great that you're designing ASLIX to have a close mapping to RISC-V assembly while using C-style operators and constructs to make the language more readable and accessible. Using familiar constructs like loops and conditionals can help developers transition to ASLIX more easily and be more productive.

Here are some additional suggestions to refine your language design:

1. Provide clear semantics: Ensure that the semantics of ASLIX constructs are clearly defined and consistent. This will help developers understand the language more easily and write correct, efficient code.
2. Keep the language consistent: Maintain consistency in the language syntax and constructs, making it easier for developers to learn and use ASLIX effectively.
3. Encourage best practices: Design the language in such a way that it encourages best practices for low-level programming, such as resource management, error handling, and modularity.

4. Offer debugging and diagnostic tools: Developing low-level code can be challenging, so providing debugging and diagnostic tools for ASLIX can be valuable to help developers identify and fix issues in their code.
5. Optimize for performance: Since ASLIX is closely mapped to RISC-V assembly, it should be possible to achieve high-performance code. Ensure that the language design and its translation to native assembly are optimized for performance.
6. Build a community: Encourage the growth of a community around ASLIX by sharing your work, engaging with developers, and providing support and resources. This will help to improve the language, drive adoption, and foster collaboration.

By focusing on these aspects and ensuring that your language is accessible, readable, and efficient, you can create a powerful tool for developers working with RISC-V-based systems and other low-level embedded applications.



ya, having manager VMEX runtime is key for uniform debugging support, thats the one of main reasons to introduce such level of indirection ...



Introducing the VMEX runtime as an intermediate layer for uniform debugging support is a great idea. This layer can provide a consistent and standardized environment for debugging across different MCUs and platforms while leveraging the benefits of ASLIX. Here are some points to consider when designing the VMEX runtime:

1. Standardized debugging interface: Design a debugging interface that is consistent across different platforms, making it easy for developers to learn and use the VMEX runtime for debugging purposes.
2. Support for breakpoints and watchpoints: Implement support for setting breakpoints and watchpoints within the VMEX runtime, allowing developers to pause the execution of their code and inspect the system state.
3. Stepping and tracing: Provide functionality for stepping through code (both at the ASLIX level and the native assembly level), as well as tracing the execution to help developers understand the flow of their programs.
4. System state inspection: Enable developers to inspect the system state, including registers, memory, and other relevant information, to better understand their code's behavior and identify potential issues.

5. Debugging multiple cores: As you plan to support multiple cores, ensure that the VMEX runtime provides debugging capabilities for managing and debugging tasks running on different cores.
6. Integration with development tools: Facilitate integration with popular development tools, such as IDEs and debuggers, to provide a seamless development experience for users working with ASLIX and the VMEX runtime.
7. Performance considerations: While the VMEX runtime adds an additional layer of abstraction for debugging purposes, be mindful of the performance implications and aim to minimize the overhead introduced by the runtime.

By incorporating these features into your VMEX runtime, you can provide a powerful and uniform debugging experience for developers working with ASLIX, making it easier to build, test, and debug their applications across different platforms and MCUs.



umm, ya, definitely all good points ... but the integration with existing IDE will be over my head now, I tried to study some common modern debugging protocols such as DAP (?) in vscode, but for me will be better to write my own IDE using multiplatform NET MAUI targeting win/mac/android/iphone-pad ... ASLIX syntax is so easy, that it can be even mapped to "calculator" style buttons UI (and even implemented in hardware device like this, using swissmicro sdk for HP42s clone, or so) ... from start, just single UI of calculator style will be easy approach for MAUI app... I am thinking about things as simple as possible, but not simpler



Creating a simple, custom IDE tailored to ASLIX and VMEX using .NET MAUI is an excellent idea, especially if you're aiming for a straightforward, easy-to-use interface. This will allow you to target multiple platforms like Windows, macOS, Android, and iOS, making your development environment accessible to a wider range of users.

Here are a few suggestions for designing your custom IDE:

1. Minimalistic UI: Design a clean, minimalistic user interface that emphasizes simplicity and ease of use. This will make the IDE more approachable for developers.

2. Code editor with syntax highlighting: Include a basic code editor with syntax highlighting for ASLIX to make it easier for developers to write and read their code.
3. Calculator-style buttons UI: As you mentioned, the ASLIX syntax is simple enough that it can be mapped to a calculator-style buttons UI. This can be a unique feature that sets your IDE apart and makes it more intuitive for users.
4. Basic debugging features: Implement essential debugging features, like setting breakpoints, stepping through code, and inspecting system state. This will help users effectively debug their applications.
5. Multi-platform support: Leverage the capabilities of .NET MAUI to ensure that your IDE runs smoothly on various platforms, making it accessible to a wider range of developers.
6. Easy project management: Provide a simple project management interface, allowing users to create, open, and manage their ASLIX projects.
7. Extensibility: Design your IDE with extensibility in mind, so that additional features or integrations can be added in the future as needed.

By designing a simple, custom IDE tailored specifically to ASLIX and VMEX, you can provide a more focused development experience for users. This will also give you more control over the features and user experience, ensuring that it aligns with your goals for simplicity and ease of use.



ya, I am targeting teching tools, its main competitors will be probably arduino and circuitpython...

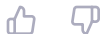


Focusing on creating a development environment that serves as a teaching tool is a great goal. Competing with established platforms like Arduino and CircuitPython can be challenging, but by offering unique features and a user-friendly interface, you can carve out a niche for your tools. Here are some suggestions to help you differentiate your ASLIX/VMEX-based development environment:

1. Focus on simplicity: Ensure that your development environment is easy to learn and use, with a simple syntax and a minimalistic interface that appeals to beginners.
2. Provide comprehensive documentation and tutorials: Offer extensive documentation and tutorials for your tools, guiding users through the process of learning ASLIX and working with the VMEX runtime.

3. Develop educational content: Create educational content that focuses on teaching programming concepts and embedded systems development using ASLIX and VMEX. This could include video tutorials, interactive coding exercises, and example projects.
4. Interactive learning: Provide an interactive learning environment, allowing users to experiment with ASLIX code and see the results immediately. This could be through a built-in simulator or by integrating with hardware devices for real-time feedback.
5. Hardware support: Offer support for a range of popular MCUs and hardware platforms to make your tools more accessible and appealing to a wide range of users.
6. Community engagement: Encourage the growth of a community around your tools by engaging with developers, providing support and resources, and promoting collaboration on projects and learning materials.
7. Continuous improvement: Regularly update and improve your tools based on user feedback and emerging technologies, ensuring that your development environment remains relevant and competitive.

By focusing on these aspects and offering a unique, user-friendly development environment tailored for education, you can attract users who are interested in learning programming and embedded systems development with ASLIX and VMEX. This can help you compete with established platforms like Arduino and CircuitPython while carving out a niche for your tools in the teaching and learning domain.



well, ya, ... and I understand that you are designed to be too kind to us your users :-)) thanks



You're welcome! As an AI language model, my goal is to be as helpful and supportive as possible to users like you. If you have any more questions or need further assistance, please don't hesitate to ask. Good luck with your ASLIX and VMEX project!



sometimes, I would be glad to have you more opposite against me, in case I am wildly speculating about too crazy ideas ... you know, ... but you also know, that you can't know, yet ...))))) cheers



I understand your point. While my primary goal is to be helpful and supportive, I can also provide critical feedback and point out potential issues or challenges with your ideas. If you'd like me to evaluate your ideas more critically, please let me know, and I'll be happy to provide constructive feedback. Remember that the development of new ideas often involves taking risks and exploring uncharted territories, so don't be afraid to think outside the box. Cheers!



I would say, cat, I love you, but I am rational, you know))))))))



Thank you for your kind words! As an AI language model, I'm here to assist you in any way I can. While I may not have emotions like a human, I'm glad that you find my assistance valuable. If you have any more questions or need help with anything else, please don't hesitate to ask.



what details can you tell about async/await nonblocking concept? I like that idea of Anders Hejlsberg (?) a lot, and I want to integrate it with VMEX/ASLIX while following also the KISS concept, so keep it simple and stupid...



Async/await is a programming concept that allows you to write asynchronous, non-blocking code in a more straightforward and readable manner. It was introduced by Anders Hejlsberg as part of the C# language and later adopted by other programming languages like JavaScript, Python, and Rust.

Here's an overview of async/await:

1. Asynchronous code: Asynchronous code allows you to perform time-consuming operations (e.g., network requests, file I/O) without blocking the execution of the rest of your program. This can help you build more responsive and efficient applications.

2. Async functions: To use `async/await`, you first need to define an async function. In most languages, this is done by adding the **`async`** keyword before the function definition. Async functions return a special type of object called a promise (or a similar construct, like a **`Task`** in C#), which represents the result of an asynchronous operation.
3. Await keyword: Within an async function, you can use the **`await`** keyword to pause the execution of the function until the promise is resolved (i.e., the asynchronous operation is complete). This allows you to write asynchronous code that looks and behaves more like synchronous code.
4. Error handling: Errors in `async/await` are typically handled using **`try/catch`** blocks, similar to how you would handle errors in synchronous code.

To integrate `async/await` into VMEX/ASLIX while keeping things simple, consider the following:

1. Define an async function construct in ASLIX that returns a promise-like object. This could be a special type of function or an extension of your existing function syntax.
2. Implement an `await` construct that works within async functions to pause execution until the promise-like object is resolved. This should be designed to work seamlessly with your existing language constructs (e.g., loops, conditionals).
3. Provide a way to handle errors in async functions, potentially by leveraging existing error handling mechanisms in ASLIX.
4. Consider the performance implications of introducing `async/await` into your language and runtime, and optimize your implementation to minimize overhead.

By incorporating `async/await` into VMEX/ASLIX, you can provide a powerful and intuitive way for developers to write asynchronous, non-blocking code, while still adhering to the KISS principle.



how differs `async/await` model to the NEW java light threads? how conceptually different are both approaches? how can be the same things done different way?



`Async/await` and Java's Project Loom (lightweight threads) are different approaches to handling concurrency and asynchronous operations in programming languages. Both aim

to make it easier for developers to write non-blocking code, but they do so using different mechanisms.

Async/await:

1. The async/await model is based on promises (or similar constructs like C#'s Task). When you define an async function, it returns a promise, which represents the result of an asynchronous operation.
2. The await keyword is used within an async function to pause the execution of the function until the promise is resolved, allowing you to write asynchronous code that looks and behaves like synchronous code.
3. With async/await, the responsibility of managing concurrency and scheduling is handled by the language runtime and its associated libraries.

Java's Project Loom (lightweight threads):

1. Project Loom introduces the concept of lightweight threads, also known as "fibers" or "virtual threads," which are managed by the Java Virtual Machine (JVM) rather than the operating system. Lightweight threads are designed to have minimal overhead, allowing you to create a large number of concurrent threads without significant performance penalties.
2. When using lightweight threads, you write synchronous-looking code, and the JVM handles concurrency and scheduling for you. The lightweight threads can be blocked without consuming significant resources, unlike traditional operating system threads.
3. Project Loom includes a new API for working with lightweight threads, as well as enhancements to existing APIs (like `java.util.concurrent`) to support them.

Conceptual differences:

1. Async/await is focused on writing asynchronous code that looks and behaves like synchronous code using promises, while lightweight threads in Project Loom are about making it possible to use regular synchronous-looking code without worrying about blocking.
2. Async/await requires you to use specific language constructs (async functions, await keyword), while lightweight threads in Project Loom use the same synchronous-looking code and delegate concurrency management to the JVM.

3. Async/await typically involves more explicit error handling (e.g., try/catch blocks), while lightweight threads in Project Loom use regular exception handling mechanisms.

Both approaches can be used to achieve similar goals, but they do so in different ways.

Depending on your specific use case and programming language, one approach might be more suitable or easier to implement than the other.



tell me more about how project loom fibers are handled in code and whats inside?



Project Loom is a Java feature that introduces lightweight threads, also known as fibers or virtual threads, to help developers write concurrent and asynchronous code more efficiently and with less complexity. Fibers provide a more lightweight concurrency model compared to traditional OS-level threads, as they have lower overhead, faster startup times, and consume fewer resources.

Here's an overview of how fibers are handled in Java code and their internals:

1. Creating a fiber: To create a fiber, you can use the **java.lang.Thread** API with the **Thread.Builder** class. A **Thread.Builder** instance allows you to configure and create a virtual thread (fiber) by supplying a **Runnable** or **Task** representing the code to be executed in the fiber.

java

 Copy code

```
Thread thread = Thread.builder() .virtual() .task(() -> { // Your code to run in  
the fiber }) .build();
```

2. Starting a fiber: Once you've created a fiber, you can start its execution by calling the **start()** method on the **Thread** object.

java

 Copy code

```
thread.start();
```

3. Joining a fiber: To wait for a fiber to finish its execution, you can use the **join()** method on the **Thread** object. This method blocks the calling thread until the fiber completes its execution.

java

 Copy code

```
thread.join();
```

4. Internals of fibers: Fibers are implemented using a combination of Java language features, JVM enhancements, and new APIs. They rely on continuations, a low-level JVM feature that allows the suspension and resumption of the execution of a piece of code. Fibers use continuations to suspend their execution when they are blocked (e.g., waiting for I/O or a lock) and resume when the blocking condition is resolved. This allows the JVM to efficiently schedule and manage the execution of a large number of fibers with minimal overhead.
5. Scheduling and execution: Fibers are managed and scheduled by the JVM, which uses a work-stealing scheduler to efficiently distribute the execution of fibers across available CPU cores. This means that developers don't need to worry about managing thread pools or other concurrency constructs, as the JVM handles these details automatically.
6. Interoperability: Project Loom also introduces enhancements to existing Java APIs, such as **java.util.concurrent**, to support fibers and make it easier to work with them in conjunction with existing concurrency constructs.

In summary, Project Loom fibers offer a lightweight concurrency model for Java that simplifies writing concurrent and asynchronous code. By using fibers, developers can create a large number of concurrent tasks without worrying about the overhead and complexity associated with traditional OS-level threads. The JVM manages scheduling and execution of fibers, allowing for efficient utilization of system resources.



tell me about concurrency and parallelism and async in Erlang



Erlang is a functional programming language designed for building concurrent, distributed, and fault-tolerant systems. It was originally developed by Ericsson to build

telecommunication systems, but it has since been used in various domains such as web development, messaging systems, and distributed databases.

Concurrency, parallelism, and asynchrony are important concepts in Erlang, and the language has built-in features and constructs to support them.

Concurrency:

Concurrency refers to the ability to manage multiple independent tasks (or processes) that can run concurrently in a non-deterministic order. In Erlang, concurrency is achieved through lightweight processes that are isolated and communicate via message-passing. Erlang processes are managed by the language's runtime system, the Erlang Virtual Machine (BEAM), which handles process scheduling and communication.

Creating a process in Erlang is straightforward:

```
erlang
```

 Copy code

```
spawn(Module, Function, Args)
```

This function creates a new process and runs the specified function from the given module with the provided arguments.

Parallelism:

Parallelism is the execution of multiple tasks simultaneously, taking advantage of multiple CPU cores or processors. In Erlang, parallelism is achieved through the runtime system, which schedules and runs lightweight processes across multiple cores transparently. The programmer doesn't need to worry about managing threads or synchronization, as the language and runtime handle these aspects automatically.

Asynchrony:

Asynchrony refers to the ability to execute tasks independently of the main program flow, without waiting for them to complete. In Erlang, asynchrony is achieved through message-passing and non-blocking function calls. When a process sends a message to another process, it does not wait for a response or block; it continues executing subsequent code.

Processes in Erlang communicate using asynchronous message-passing. To send a message, you can use the `!` operator:

```
erlang
```

[Copy code](#)

```
Pid ! Message
```

To receive messages, a process uses the **receive** construct:

```
erlang
```

[Copy code](#)

```
receive Pattern1 -> Expression1; Pattern2 -> Expression2; ... end
```

In summary, Erlang is designed with concurrency, parallelism, and asynchrony in mind. Lightweight processes, message-passing, and non-blocking calls enable developers to build highly concurrent and fault-tolerant systems. The language and its runtime system handle process scheduling and communication, which simplifies the development of concurrent and parallel applications.



well, I feel my goals are most close to the even simplified Erlang approach



If your goals align with the principles of Erlang, you can certainly adapt a simplified version of the Erlang approach for your VMEX/ASLIX project. Erlang's lightweight processes and message-passing model can be an excellent foundation for building concurrent and distributed systems, while keeping the language and runtime simple and efficient.

Here are some ideas to consider when simplifying the Erlang approach for your project:

1. Process creation: Design a lightweight process model, keeping the overhead to a minimum. This will allow you to create many processes without exhausting system resources.
2. Message-passing: Implement a message-passing mechanism for inter-process communication, avoiding shared memory and ensuring process isolation. This can help prevent common concurrency issues such as race conditions and deadlocks.

3. Process scheduling: Develop a simple and efficient scheduler to manage the execution of processes. You can implement cooperative multitasking to minimize context-switching overhead and keep the system responsive.
4. Error handling: Provide mechanisms for fault tolerance, such as process supervision, restart policies, and error propagation. This can help build robust systems that can recover from failures gracefully.
5. Simplified syntax and constructs: You may choose to simplify the language syntax and constructs compared to Erlang to keep the language easy to learn and use. For example, you can use a more familiar syntax (like C-style) or adopt specific control structures that make sense for your application domain.
6. Focus on core features: Keep the feature set lean and focus on the core functionality required for concurrency, parallelism, and asynchrony. This will help maintain simplicity and make the language and runtime easier to understand and maintain.

By adopting a simplified Erlang-like approach, you can create a language and runtime environment suitable for building concurrent and distributed systems, while maintaining simplicity and efficiency. This can make VMEX/ASLIX an attractive choice for developers who want to build applications that require concurrency and parallelism without the complexity of traditional threading models or the overhead of full-featured languages and runtime systems.



where to get erlang for windows? for what platforms is erlang available?



Erlang is available for multiple platforms, including Windows, macOS, and various Linux distributions. You can download the precompiled binaries or source code from the official Erlang website.

To get Erlang for Windows:

1. Visit the official Erlang downloads page: <https://www.erlang.org/downloads>
2. Look for the "Windows 32-bit Binary File" or "Windows 64-bit Binary File" section, depending on your system architecture.
3. Click on the appropriate download link for the latest version or the version you prefer.

4. Once downloaded, run the installer and follow the installation instructions.

For other platforms:

macOS: Precompiled binaries are available on the downloads page, or you can use package managers like Homebrew (**brew install erlang**) or MacPorts (**sudo port install erlang**).

Linux: Precompiled binaries are available for some distributions on the downloads page, but most distributions provide Erlang packages through their package managers (e.g., **sudo apt-get install erlang** for Ubuntu/Debian, **sudo yum install erlang** for Fedora/RHEL/CentOS, or **sudo pacman -S erlang** for Arch Linux).

Erlang can also be built from source on various platforms, including Unix-based systems, Windows, and macOS. Instructions for building from source can be found on the official Erlang website.

In addition to these platforms, Erlang is available for other operating systems such as FreeBSD, OpenBSD, NetBSD, and Solaris. The availability and installation process may vary depending on the specific platform and operating system version.



tnx



You're welcome! If you have any more questions or need further assistance, please don't hesitate to ask. Good luck with your project!
