



(2303326c) VMEX = Virtual Multicore Executive runtime with translated RV32E for (tiniest) MCUs. We will be designing the core runtime features along with interpreter.



It sounds like you're working on a project to create a Virtual Multicore Executive (VMEX) runtime environment, designed to utilize translated RISC-V 32-bit embedded (RV32E) instruction set architecture for small microcontroller units (MCUs). To help you get started, I've outlined the core runtime features and steps to develop an interpreter.

Core runtime features:

1. Task scheduler: Implement a task scheduler to manage the execution of multiple tasks concurrently. The scheduler should support priority-based or round-robin scheduling, depending on the use case.
2. Inter-process communication: Provide a mechanism for tasks to communicate and synchronize with each other. This could be through message passing, shared memory, or a combination of both.
3. Resource management: Design resource management functionality to handle allocation and deallocation of memory and other system resources for tasks.
4. Fault handling: Implement fault detection and recovery mechanisms to ensure system stability and reliability. This could include error handling, exception handling, and watchdog timers.
5. Power management: Provide power management features that allow tasks to be executed with minimal energy consumption. This is especially important for MCUs used in low-power applications.

Developing an interpreter:

1. Define the bytecode: Create a bytecode representation for the translated RV32E instructions. This will enable efficient interpretation and execution of the instructions on the MCU.
2. Parsing and decoding: Implement a parser to read the input bytecode and decode it into the corresponding RV32E instructions.

3. Execution engine: Design an execution engine that takes decoded instructions and executes them on the virtual multicore runtime. This includes maintaining the state of the virtual cores and handling inter-core communication and synchronization.
4. Integration: Integrate the interpreter with the VMEX runtime, ensuring seamless interaction between the tasks running on the virtual multicore environment and the underlying MCU hardware.
5. Optimization: Optimize the interpreter for speed and resource efficiency, which is crucial for tiny MCUs. Techniques such as Just-In-Time (JIT) compilation, caching, or loop unrolling can be used to improve performance.
6. Testing and debugging: Develop a testing framework to ensure the correctness and robustness of the interpreter, as well as to identify and fix any issues.

Once you have designed and implemented the core runtime features and the interpreter, thorough testing and optimization will be necessary to ensure efficient and reliable operation on tiny MCUs.



Definitely good points. My primary goal is single unified ISA for all MCUs around, so also 8bit, 16bit natively, executing 1:1 translated RV32E into more optimized binary encoding for sw interpretation with C or assembly implementation. Another goal is multicore with separated memory areas and simple transparent task communication with the core runtime. Core runtime also must provide bare-metal access to serial uart terminal and provide access to programming flash memory, having also simple interactive monitor over the terminal.



It's great that you have a clear set of goals for your project. Based on your requirements, I'll outline an approach to achieve them:

1. Unified ISA for all MCUs:

Design a custom binary encoding format that can represent the RV32E instructions more efficiently for software interpretation. This format should be compatible with 8-bit and 16-bit MCUs, allowing them to execute the translated RV32E code natively. Develop a translation tool that can convert RV32E instructions into the custom binary encoding format. This tool can be used to translate RV32E programs for execution on different MCU architectures.

2. Multicore support with separated memory areas:

Design the VMEX runtime to support multiple cores, each with their own dedicated memory areas.

Implement memory protection mechanisms to ensure that tasks running on different cores cannot accidentally access each other's memory.

3. Transparent task communication:

Develop a simple, transparent communication mechanism for tasks running on different cores. This could be achieved using message passing, shared memory buffers, or a combination of both.

Design APIs for tasks to easily send and receive data between cores.

4. Bare-metal access to UART and flash memory:

Implement low-level drivers to provide access to the MCU's UART for serial communication, and flash memory for program storage.

Develop APIs for tasks to read/write data from/to the UART and flash memory.

5. Interactive monitor over the terminal:

Design an interactive monitor application that runs on the MCU and communicates with the user via the UART terminal.

Implement monitor commands for tasks to inspect and control the VMEX runtime, such as listing active tasks, starting/stopping tasks, and modifying memory.

6. C and Assembly implementation:

Develop the core runtime and interpreter in C and/or assembly language to ensure the best performance and resource utilization on the target MCUs.

After you've designed and implemented the aforementioned components, thorough testing and optimization will be necessary to ensure efficient and reliable operation on a wide range of MCUs. By addressing these goals, you'll be able to create a flexible and efficient VMEX runtime environment that can effectively execute translated RV32E code on various MCU architectures, while providing multicore support and simple task communication.



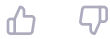
Well, VMEX interpreter binary encoding is already defined, I have also with your help implemented basic interpreter in C in 2kB of code on 8bit AVR, thats okay for now. Now I am thinking loudly about the most important core features, ya. Goal is also to fit entire runtime into 8kB of flash code, enabling at least 16kB MCUs to leave some room for application code and having enough RAM for at least few virtual cores running.



It's great to hear that you've made progress on the VMEX interpreter binary encoding and have a working implementation on an 8-bit AVR MCU. Given the constraints of fitting the entire runtime into 8 kB of flash memory and leaving enough RAM for multiple virtual cores, I will provide a list of essential core features to focus on:

1. **Lightweight task scheduler:** Implement a minimal task scheduler that efficiently manages the execution of tasks on virtual cores. It should be able to handle context switching, and you may choose a simple scheduling algorithm, such as round-robin or cooperative multitasking.
2. **Memory-efficient inter-process communication:** Design a communication mechanism for tasks that uses minimal memory overhead. A shared memory buffer or a simple message-passing system using fixed-size data structures would be suitable options.
3. **Resource management:** Implement a basic resource management system that takes care of memory allocation and deallocation for tasks. You can use a lightweight memory allocator or even rely on fixed-size memory pools to keep the implementation size small.
4. **Bare-metal UART and flash memory access:** Provide simple APIs for tasks to read/write data from/to the UART and flash memory. This can be achieved using low-level drivers that interface directly with the MCU's hardware registers.
5. **Core runtime and interpreter optimizations:** Optimize the core runtime and interpreter code for size and performance. Techniques such as function inlining, dead code elimination, and loop unrolling can help reduce the code size and improve execution speed.
6. **Minimal error handling and fault recovery:** Implement basic error handling and fault recovery mechanisms to ensure system stability. You can opt for simple error reporting and task restarts in case of faults.
7. **Configurability:** Design the VMEX runtime to be easily configurable, allowing users to enable or disable specific features based on their requirements. This can help reduce the code size and memory usage, enabling the runtime to fit within the 8 kB constraint.

With these essential core features, you should be able to create a minimal and efficient VMEX runtime that fits within the constraints of 8 kB flash memory and leaves enough room for application code and RAM for multiple virtual cores. Remember to focus on simplicity and efficiency in your implementation to make the best use of the limited resources available on small MCUs.



can you provide some routines to program flash on AVR? how these differs between generations (ATmega, ATtiny, ...)



AVR microcontrollers, such as ATmega and ATtiny series, have a similar process for programming flash memory. While there might be minor differences in register names or memory sizes, the basic concept remains the same. Here's an example of a simple routine to program flash on AVR MCUs:

1. Include necessary headers:

c

Copy code

```
#include <avr/io.h> #include <avr/pgmspace.h> #include <avr/interrupt.h>
```

2. Define a function to enable and wait for the completion of flash writing:

c

Copy code

```
void waitForFlashReady() { while (NVM_STATUS & (1 << NVM_NVMBUSY_bp)); }
```

3. Define a function to write a page buffer to flash memory:

c

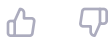
Copy code

```
void writeFlashPage(uint32_t pageAddress, uint8_t *data, uint16_t pageSize) {  
    uint16_t i; // Disable interrupts cli(); // Erase the target flash page  
    waitForFlashReady(); NVM_CMD = NVM_CMD_ERASE_APP_PAGE_gc; CCP = CCP_SPM_gc;  
    NVM_EXEC(); // Write data to the page buffer for (i = 0; i < pageSize; i += 2) {  
        uint16_t word = data[i] | (data[i + 1] << 8); waitForFlashReady(); NVM_CMD =  
        NVM_CMD_LOAD_FLASH_BUFFER_gc; CCP = CCP_SPM_gc; *((uint16_t *) (pageAddress + i))  
        = word; } // Write the page buffer to flash memory waitForFlashReady(); NVM_CMD  
        = NVM_CMD_WRITE_APP_PAGE_gc; CCP = CCP_SPM_gc; NVM_EXEC(); // Re-enable  
        interrupts sei(); }
```

In the example above, replace `NVM_STATUS`, `NVM_NVMBUSY_bp`, `NVM_CMD`, `NVM_CMD_ERASE_APP_PAGE_gc`, `NVM_CMD_LOAD_FLASH_BUFFER_gc`, `NVM_CMD_WRITE_APP_PAGE_gc`, `CCP`, and `CCP_SPM_gc` with the appropriate register and bit names for your specific AVR MCU.

The differences between AVR generations (ATmega, ATtiny, etc.) mostly lie in the register names, memory sizes, and available features. You may need to refer to the respective datasheet for the correct register names and other specifics for the microcontroller you're using. The overall process of programming flash memory remains the same, but make sure to adapt the example provided to your specific MCU.

It's essential to consult the microcontroller's datasheet to understand any variations between different AVR generations and models. The datasheet will provide you with the correct register names, memory sizes, and other information you need to adapt the code for your specific microcontroller.



can be also thing like OPTIBOOT loader used for the flashing?



Yes, Optiboot is an excellent choice for a bootloader for flashing AVR microcontrollers. Optiboot is a small, efficient, and widely-used bootloader that is compatible with various AVR MCUs, including ATmega and ATtiny series. It is the default bootloader for the Arduino UNO and other Arduino boards.

To use Optiboot for flashing, follow these steps:

1. Obtain the Optiboot source code and compile it for your specific AVR microcontroller. The Optiboot GitHub repository (<https://github.com/Optiboot/optiboot>) provides the source code and build instructions. Make sure to select the appropriate configuration for your target MCU and clock speed.
2. Flash the Optiboot bootloader to your AVR microcontroller using a hardware programmer (such as USBasp, AVRISP mkII, or Arduino as ISP). The programmer will write the bootloader to the last pages of the flash memory, leaving the rest for your application code.

3. Configure the fuses on your AVR microcontroller according to the Optiboot documentation. The fuses control various settings such as the boot section size, boot reset vector, and brown-out detection level.
4. Modify your application code to be compatible with Optiboot. Your application code should not overwrite the bootloader section of the flash memory, and it should start at the bootloader's designated entry point (typically 0x0000 for Optiboot). Ensure that your application's linker script or build settings are configured accordingly.
5. Connect your AVR microcontroller to a UART interface (such as an FTDI adapter or a USB-to-serial converter) for communication with your PC. Optiboot uses the UART interface to receive new firmware and flash it to the microcontroller.
6. Use a tool like **avrdude** to upload your application code to the microcontroller through the Optiboot bootloader. **avrdude** supports various programming protocols, including the STK500 protocol used by Optiboot. An example command to upload firmware using **avrdude** is:

CSS

 Copy code

```
avrdude -c arduino -p <MCU_NAME> -P <SERIAL_PORT> -b 115200 -U flash:w:
<APPLICATION_HEX_FILE>
```

Replace **<MCU_NAME>** with your specific AVR microcontroller (e.g., **atmega328p**), **<SERIAL_PORT>** with the appropriate serial port on your PC (e.g., **/dev/ttyUSB0** on Linux, **COM3** on Windows), and **<APPLICATION_HEX_FILE>** with the path to your compiled application code in Intel HEX format.

By following these steps, you can use the Optiboot bootloader to flash your AVR microcontroller. This approach eliminates the need for a dedicated hardware programmer for each firmware update and makes it easier to deploy new firmware to your devices.



what is godolt compiler erxplorer?



It seems like you meant "Godbolt Compiler Explorer." The Godbolt Compiler Explorer is an interactive web-based tool that allows users to compile code in various programming

languages (such as C, C++, Rust, and others) using different compilers and compiler options. It displays the resulting assembly or machine code output side-by-side with the original source code, making it easy to analyze and understand the generated code.

The Compiler Explorer, created by Matt Godbolt, is particularly useful for:

1. Analyzing the performance and efficiency of your code at the assembly level.
2. Comparing the output of different compilers or different versions of the same compiler.
3. Understanding how compiler optimizations affect the generated code.
4. Learning about assembly language and the inner workings of compilers.

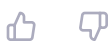
The Godbolt Compiler Explorer supports a wide range of compilers, including GCC, Clang, MSVC, and others, as well as various target architectures like x86, ARM, RISC-V, and more.

You can access the Godbolt Compiler Explorer at: <https://godbolt.org/>

To use the Compiler Explorer:

1. Write or paste your source code in the left pane.
2. Choose the desired compiler, target architecture, and compiler options from the top menu.
3. View the generated assembly or machine code in the right pane, with corresponding source code lines highlighted for easy comparison.

Compiler Explorer is an invaluable resource for developers who want to gain deeper insights into how their code is compiled and executed on different hardware platforms and compiler configurations.



do you know internal structure of optiboot loader?



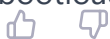
Optiboot is a lightweight and efficient bootloader for AVR microcontrollers. Its primary purpose is to receive firmware updates over a UART interface, program the firmware into the microcontroller's flash memory, and launch the application code.

The internal structure of Optiboot can be divided into several key components:

1. Initialization: Optiboot starts by initializing the hardware and setting up the UART interface for communication. This includes configuring the baud rate, data bits, stop bits, and parity settings according to the specific AVR microcontroller being used.
2. Watchdog timer: Optiboot utilizes the watchdog timer to automatically reset the microcontroller after a predefined time interval if no valid firmware update is received. This ensures that the bootloader does not indefinitely wait for new firmware, allowing the existing application code to run in case of communication issues or other errors.
3. Firmware reception: Optiboot listens for incoming firmware updates over the UART interface. It implements the STK500 protocol, a widely-used communication protocol for programming AVR microcontrollers. The bootloader parses the received commands, checks their validity, and stores the incoming firmware data in a buffer.
4. Flash programming: After receiving a valid firmware update, Optiboot writes the new firmware data to the microcontroller's flash memory. It uses the AVR's self-programming capability, which allows the microcontroller to write to its own flash memory under specific conditions.
5. Application launch: Once the firmware update is complete, Optiboot jumps to the application code's entry point, effectively launching the updated application. Typically, the bootloader is located in the upper part of the flash memory, while the application code resides in the lower part.
6. Error handling: Optiboot incorporates basic error handling to ensure that firmware updates are processed correctly. It can detect errors such as invalid commands, incorrect data sizes, or flash write failures. In case of an error, the bootloader sends an error response over the UART interface and resets the microcontroller to retry or launch the existing application code.

The Optiboot bootloader is designed to be small and efficient, minimizing the amount of flash memory and RAM used. Its source code is available on GitHub

(<https://github.com/Optiboot/optiboot>), allowing developers to modify and customize the bootloader according to their specific requirements.



thanks, you are good buddy and intellient colleague :-> tell me please more about STK500 protocol



Thank you for your kind words! I'm glad to be of assistance.

STK500 is a communication protocol developed by Atmel (now part of Microchip) for programming their AVR microcontrollers. The protocol is named after the Atmel STK500 development board, which was one of the first boards to use this protocol for communication between the host computer and the AVR microcontroller.

The STK500 protocol uses a simple command-response structure over a serial (UART) interface to facilitate communication. It supports various operations, including reading and writing flash memory, reading and writing EEPROM, setting fuses, and locking/unlocking the microcontroller.

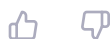
Here's an overview of the STK500 protocol:

1. Packet structure: The STK500 protocol consists of packets exchanged between the host computer and the microcontroller. Each packet begins with a start byte (0x1B), followed by a sequence number, a message size, a token (0x0E), the actual command or data bytes, and a checksum.
2. Command set: The STK500 protocol defines a set of commands for various operations. Some common commands include:

CMD_SPI_MULTI: Perform multiple SPI operations.

CMD_ENTER_PROGMODE_ISP: Enter programming mode using the ISP interface.

CMD_LEAVE_PROGMODE_ISP: Leave



okay, can be stk500 protocol abstracted to be used on multiple different MCUs? also PIC16, PIC18, PIC24 and ARM M0+?



Yes, the STK500 protocol can be abstracted and adapted for use with multiple different microcontrollers, including PIC16, PIC18, PIC24, and ARM Cortex-M0+ MCUs. However, you would need to create custom bootloader firmware for each microcontroller that implements

the STK500 protocol and handles the specific programming requirements of the target MCU.

To implement the STK500 protocol on different microcontrollers, follow these general steps:

1. Familiarize yourself with the target microcontroller's programming method, memory layout, and any specific programming requirements as described in the respective datasheets and reference manuals.
2. Develop a bootloader for the target microcontroller that initializes the hardware, sets up the UART interface for communication, and waits for incoming STK500 packets.
3. Implement the STK500 packet parsing and processing in the bootloader, extracting command and data information from the received packets.
4. Adapt the bootloader to handle the specific memory programming operations (such as flash and EEPROM read/write, fuse settings, etc.) for the target microcontroller. This might involve writing custom functions for erasing, writing, and reading memory, as well as handling any special requirements for the target MCU.
5. Implement error handling and responses according to the STK500 protocol, providing feedback to the host computer about the success or failure of the requested operations.
6. Test the bootloader with the target microcontroller, ensuring that it can correctly program the device using the STK500 protocol. You may need to use a programming tool like **avrdude** with a custom configuration to support the target MCU.

By following these steps, you can create a bootloader that implements the STK500 protocol for various microcontrollers. Keep in mind that each microcontroller family may have different programming requirements and hardware interfaces, so thorough understanding and adaptation of the bootloader code is essential for successful implementation.

👍 👎



well, exist some STK500 similar absolute minimum protocol for firmware updating?



Yes, you can design a minimalistic custom protocol for firmware updating that is simpler than the STK500 protocol. The primary goal of such a protocol is to provide basic functionality for firmware reception and flash programming.

Here's an outline of an absolute minimum protocol for firmware updating:

1. Packet structure: Define a simple packet structure that includes a start byte, packet type (e.g., command, data, or end of transmission), payload length, payload data (if applicable), and a checksum or CRC for error detection.
2. Command set: Define a minimal command set for the required operations. For example:
ENTER_PROGRAMMING_MODE: Enter programming mode and prepare the microcontroller for firmware updating.
DATA_PACKET: Receive a chunk of firmware data.
EXIT_PROGRAMMING_MODE: Exit programming mode and finalize the firmware update.
3. Firmware reception: The bootloader listens for incoming packets over the UART interface, parses the packet structure, and processes the commands. For data packets, the bootloader stores the received data in a buffer.
4. Flash programming: Once the bootloader has received a complete firmware update, it writes the new firmware data to the microcontroller's flash memory. This process may involve erasing and writing individual pages or sectors, depending on the target microcontroller's architecture.
5. Application launch: After successfully updating the firmware, the



well, what about integrating such firmware updating protokol into tiny interactive monitor, using some simple hex ascii transfer protocol? which is simplest? intel hex? motorola s? or soomething simpler?



Integrating a simple firmware updating protocol into a tiny interactive monitor using an ASCII-based transfer protocol is a good approach to minimize complexity and resource usage. Among the common ASCII-based hex transfer protocols, Intel HEX and Motorola S-Record are popular choices. However, for an even simpler implementation, you can create your custom hex transfer protocol.

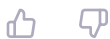
Here's a minimal custom hex transfer protocol for firmware updating:

1. Packet structure: Define a basic packet structure that consists of a start character, an address, a data length, the payload data, and a checksum or CRC for error detection. For example:

```
:start_address:data_length:data:checksum
```

2. Packet reception: The interactive monitor listens for incoming packets over the UART interface. When a packet is received, it extracts the address, data length, payload data, and checksum. Then, it verifies the checksum to ensure data integrity.
3. Memory buffer: Store the received payload data in a buffer, using the specified address as an index. Continue buffering the received data until the entire firmware update is received.
4. Flash programming: Once the entire firmware update is received, the interactive monitor writes the new firmware data to the microcontroller's flash memory. This process may involve erasing and writing individual pages or sectors, depending on the target microcontroller's architecture.
5. Application launch: After successfully updating the firmware, the interactive monitor jumps to the application code's entry point, effectively launching the updated application.

By using a custom minimal hex transfer protocol, you can reduce complexity and resource usage while still providing essential functionality for firmware updating. This approach is well-suited for tiny interactive monitors running on resource-constrained microcontrollers.



okay, can you provide some simple C implementation of hex ascii protocol as interactive monitor comamnds over serial uart?



Here's a simple implementation of a hex ASCII protocol for an interactive monitor over a UART interface. This example assumes that you have already set up the UART and have functions for sending and receiving characters.

```
#include <stdint.h> #include <stdbool.h> #include <string.h> #include <ctype.h>
#define BUFFER_SIZE 256 uint8_t firmware_buffer[BUFFER_SIZE]; uint8_t
```

```

read_hex_digit(char ch) { if (ch >= '0' && ch <= '9') { return ch - '0'; } else
if (ch >= 'A' && ch <= 'F') { return ch - 'A' + 10; } else if (ch >= 'a' && ch
<= 'f') { return ch - 'a' + 10; } return 0; } uint8_t read_hex_byte(char *str) {
return (read_hex_digit(str[0]) << 4) | read_hex_digit(str[1]); } bool
process_hex_packet(char *packet) { if (packet[0] != ':') { return false; //
Invalid start character } uint16_t address = (read_hex_byte(packet + 1) << 8) |
read_hex_byte(packet + 3); uint8_t data_length = read_hex_byte(packet + 5); if
(address + data_length > BUFFER_SIZE) { return false; // Out of buffer bounds }
uint8_t checksum = 0; for (int i = 0; i < data_length; ++i) { uint8_t data_byte
= read_hex_byte(packet + 7 + i * 2); firmware_buffer[address + i] = data_byte;
checksum += data_byte; } uint8_t received_checksum = read_hex_byte(packet + 7 +
data_length * 2); return checksum == received_checksum; } void
interactive_monitor() { char packet[64]; size_t packet_index = 0; while (1) {
char ch = uart_receive_char(); if (ch == '\n' || ch == '\r') { if (packet_index
> 0) { packet[packet_index] = '\0'; bool success = process_hex_packet(packet);
uart_send_string(success ? "OK\n" : "ERROR\n"); packet_index = 0; } } else if
(packet_index < sizeof(packet) - 1 && (isxdigit(ch) || ch == ':')) {
packet[packet_index++] = ch; } else { // Invalid character, reset packet
packet_index = 0; } } }

```

This example provides a simple implementation of a custom hex ASCII protocol. The **process_hex_packet()** function processes the received hex packets, stores the data in the **firmware_buffer**, and checks the checksum. The **interactive_monitor()** function reads incoming characters from the UART, assembles the hex packets, and processes them.

Please note that this example does not include the flash programming and application launch steps, which depend on your

network error

