



uses risc-v some pseudoinstructions for push/pull to/from stack?



>

RISC-V is an open-source instruction set architecture (ISA) that uses a reduced instruction set, making it simple and efficient. While RISC-V does not have explicit "push" and "push" instructions like some other ISAs, it is possible to use a combination of existing instructions to perform stack push and pull operations.

A stack in RISC-V typically grows downwards, meaning that as new elements are pushed onto the stack, the stack pointer (usually the x2 register, also known as "sp") is decremented.

Here's an example of how you can implement a push and pull operation using RISC-V assembly:

To push a register onto the stack:

bash

Copy code

```
addi sp, sp, -8 # Decrement stack pointer by 8 bytes (doubleword) sd rx, 0(sp) #  
Store the value of register 'rx' at the address pointed to by the stack pointer
```

To pull a register from the stack:

bash

Copy code

```
ld rx, 0(sp) # Load the value at the address pointed to by the stack pointer  
into register 'rx' addi sp, sp, 8 # Increment the stack pointer by 8 bytes  
(doubleword)
```

Replace 'rx' with the register you want to push or pull (e.g., x1, x5, x31, etc.). Note that these examples assume a RISC-V 64-bit implementation, as we are using 8-byte doublewords. For a 32-bit implementation, you would use 4-byte words and adjust the instructions accordingly (e.g., using 'sw' and 'lw' instead of 'sd' and 'ld', and updating the immediate values in the 'addi' instructions).



well, I expect that pushing / pulling stack in riscv happens usually on more registers at once, ya? and how is calling of subroutines with pushing return address to stack handled?



Yes, you're correct. In many cases, pushing or pulling multiple registers to/from the stack is done at once, especially during function calls or when using callee-saved registers. This is to

ensure that the current state of registers is preserved before executing the called function and then restored after returning from the function.

In RISC-V, the return address is stored in the x1 register (also known as "ra"). When a function is called, the 'jal' (jump and link) instruction is used, which jumps to the target address and stores the return address in the x1 register. To push the return address to the stack and call a function, you can use the following assembly code:

bash

 Copy code

```
# Save the return address (x1) and any other registers you need to preserve
addi sp, sp, -16 # Decrement the stack pointer by 16 bytes (assuming a 64-bit
implementation)
sd ra, 0(sp) # Save the return address (x1) on the stack
sd rx, 8(sp) # Save another register (e.g., x5, x31) on the stack, if needed
jal ra, target_function # Jump to the target function and store the return address
in the x1 register
# Restore the saved registers and return
ld rx, 8(sp) # Restore the previously saved register
ld ra, 0(sp) # Restore the return address
addi sp, sp, 16 # Increment the stack pointer by 16 bytes
jr ra # Jump to the return address (stored in x1)
```

Replace 'rx' with the register you want to save/restore (e.g., x5, x31, etc.). If you need to save/restore multiple registers, adjust the stack pointer decrement/increment and add more 'sd' and 'ld' instructions accordingly.

Please note that this example assumes a RISC-V 64-bit implementation. For a 32-bit implementation, you would use 4-byte words and adjust the instructions accordingly (e.g., using 'sw' and 'lw' instead of 'sd' and 'ld', and updating the immediate values in the 'addi' instructions).



for this board W65C265SXB SBC, which are its monitor commands?



The W65C265SXB is a single-board computer (SBC) based on the W65C816S microprocessor, which is a 16-bit enhancement of the popular 8-bit 6502 microprocessor. The W65C265SXB includes an on-board monitor program called the W65C816SXB Monitor, which provides users with a command-line interface to interact with the system.

While I couldn't find a comprehensive list of monitor commands specifically for the W65C265SXB SBC, I can provide you with a list of common monitor commands that are often available on systems based on the W65C816S microprocessor.

1. Memory manipulation:

D (Display memory): Display the contents of a memory range.

M (Modify memory): Modify the contents of a memory location.

F (Fill memory): Fill a range of memory locations with a specific value.

C (Compare memory): Compare two memory ranges.

2. Register manipulation:

R (Read registers): Display the contents of the CPU registers.

S (Set registers): Set the values of the CPU registers.

3. Program execution:

G (Go): Execute a program at a specified address.

T (Trace): Execute one instruction and display the CPU registers.

BP (Breakpoint): Set, remove, or display breakpoints.

BR (Break on Register): Break on a register access.

4. Load/save programs:

L (Load): Load a program from storage (e.g., a disk, tape, or serial port) into memory.

X (Xmodem): Transfer a file using the Xmodem protocol.

S (Save): Save a program from memory to storage.

5. I/O and system commands:

I (Input): Read a value from an I/O port.

O (Output): Write a value to an I/O port.

H (Help): Display a list of available commands.

V (Version): Display the monitor version information.

Please note that the availability and syntax of these commands may vary depending on the specific monitor program used on the W65C265SXB SBC. You should consult the documentation or source code of the monitor program to get the exact list of supported commands and their usage.



can you list me example terminal code using cc65 tgi on commodore plus4



The cc65 is a C cross-compiler suite for 6502-based systems, including the Commodore Plus/4. The suite contains a C compiler, assembler, linker, and several libraries, including the TGI (Tiny Graphics Interface) library, which provides a unified, high-level graphics API for various target platforms.

Here's an example of a simple program that uses the cc65 TGI library to draw a few shapes on the screen of a Commodore Plus/4:

c

Copy code

```
#include <tgi.h> #include <conio.h> // Declare TGI driver for Plus/4 extern char
plus4tgi[]; void main(void) { // Install and initialize the TGI driver
tgi_install(plus4tgi); tgi_init(); while (tgi_busy()); // Wait for the driver to
finish initializing tgi_clear(); // Clear the screen // Draw a line
tgi_setcolor(TGI_COLOR_WHITE); tgi_line(20, 20, 100, 20); // Draw a rectangle
tgi_setcolor(TGI_COLOR_CYAN); tgi_rectangle(30, 30, 80, 50); // Draw a circle
tgi_setcolor(TGI_COLOR_YELLOW); tgi_circle(100, 100, 20); tgi_updatedisplay();
// Refresh the screen // Wait for a keypress to exit cgetc(); // Deinitialize
TGI and uninstall the driver tgi_deinit(); tgi_uninstall(); }
```

To compile and link this program for the Commodore Plus/4, you need to install the cc65 compiler suite, and then use the following commands in the terminal:

mathematica

 Copy code

```
cc65 -t plus4 -O -Or -Cl -Os example.c -o example.s ca65 example.s -o example.o
ld65 -C plus4-stdser.cfg -o example.prg example.o plus4.lib
```

This will generate a file called **example.prg**, which you can then load and run on a Commodore Plus/4 or an emulator.

Please note that the actual TGI driver for Plus/4 might have a different name than **plus4tgi** in the cc65 library. Consult the cc65 documentation for the exact name of the driver for your target platform.



well, I am more interested in text console example terminal program



If you're interested in a text console example for the Commodore Plus/4 using the cc65 suite, you can create a simple program that takes user input and prints it on the screen. Here's an example:

c

 Copy code

```
#include <conio.h> #include <stdio.h> void main(void) { char input; // Set up
the text screen clrscr(); bgcolor(COLOR_BLACK); textcolor(COLOR_WHITE);
cursor(1); // Print an introduction message cputs("Commodore Plus/4
Terminal\r\n"); cputs("=====\r\n"); cputs("Type anything and
press Enter.\r\n"); cputs("Type 'q' and press Enter to quit.\r\n"); cputs("\r\n>
"); // Main input loop while (1) { input = cgetc(); // Wait for a key press
cputc(input); // Print the entered character // Check for the Enter key or 'q'
key if (input == '\r') { cputs("\r\n> "); } else if (input == 'q') { break; } }
// Clear the screen and print an exit message clrscr(); cputs("Goodbye!\r\n"); }
```

To compile and link this program for the Commodore Plus/4, you need to install the cc65 compiler suite, and then use the following commands in the terminal:

mathematica

 Copy code

```
cc65 -t plus4 -O -Or -Cl -Os example.c -o example.s ca65 example.s -o example.o
ld65 -C plus4-stdser.cfg -o example.prg example.o plus4.lib
```

This will generate a file called **example.prg**, which you can then load and run on a Commodore Plus/4 or an emulator. The program will display a simple text console, allowing

you to type and display characters. Pressing 'q' and Enter will exit the program.



what is content of flash memory when page is cleared?



When a page of flash memory is erased, its content is typically filled with a default value. For most flash memory technologies, the erased state corresponds to all bits set to 1, or in other words, each byte in the erased page contains the value 0xFF (255 in decimal).

In a nutshell, when a page of flash memory is cleared or erased, its content becomes all 1s, or 0xFF for each byte in the page.



Are there some C headers for peripheral registers of W65C265S chip?



As of my knowledge cutoff date in September 2021, there isn't a widely-used or official C header file specifically for the W65C265S microcontroller peripheral registers. However, you can create your own C header file that defines the peripheral registers and their addresses based on the W65C265S data sheet.

Here's an example of a simple header file defining a few peripheral registers of the W65C265S chip:

c

Copy code

```
#ifndef W65C265S_H #define W65C265S_H #include <stdint.h> // Memory-Mapped I/O
Base Address for W65C265S Peripherals #define IO_BASE 0x0000 // UART1 Registers
#define UART1_BASE (IO_BASE + 0x1C00) typedef volatile struct { uint8_t
U1_RBR_THR_DLL; // Receive Buffer Register (RBR) / Transmit Holding Register
(THR) / Divisor Latch (Low) uint8_t U1_IER_DLH; // Interrupt Enable Register
(IER) / Divisor Latch (High) uint8_t U1_IIR_FCR; // Interrupt Identification
Register (IIR) / FIFO Control Register (FCR) uint8_t U1_LCR; // Line Control
Register uint8_t U1_MCR; // Modem Control Register uint8_t U1_LSR; // Line
Status Register uint8_t U1_MSR; // Modem Status Register uint8_t U1_SCR; //
Scratch Register } W65C265S_UART1_TypeDef; #define UART1
((W65C265S_UART1_TypeDef *)UART1_BASE) // ... Add other peripheral register
definitions here #endif // W65C265S_H
```

This example defines a structure for UART1 peripheral registers and assigns it to the appropriate memory-mapped address. You can expand this header file to include other peripheral register definitions based on the W65C265S data sheet.

To use this header file in your C code, simply include it like any other header file:

```
#include "w65c265s.h" int main(void) { // Example: Set the UART1 Line Control
Register UART1->U1_LCR = 0x03; // ... Add your application code here return 0; }
```

Keep in mind that this header file is just an example, and you may need to adjust or expand it based on your specific requirements and the W65C265S data sheet.



can you show me template code for simple monitor commands for this VMEX emulator implemented in vmex() function and around Core data structure? I will be probably changing that structure to something more generic later, having only _prog_ and _data_ pointers stored in Core frame along with registers and PC ... this will be specified later more, but for now, I am interested in simple interactive repl using ">" as prompt and "r" for registers content listing on single core.

```

#define RUN_TESTS
#define PAGE_SIZE (4*32) //16 at least for tests (16 instructions test scripts)
#define PAGE_SIZE (4 * 8) //8 for hypothetical tests on smallest 512byte RAM an 4kb FLASH
PIC16F

#define CONSOLE //not yet supported but will be part of the small/classic/aslix monitor
//goal is to support tiniest serial terminal but also allow larger ANSI/VT100-
like
//or at least CPlus/4-like PETSCII) terminal with control cursor commands
(escape sequences).
//on embedded hardware the connection is GPIO powered SW UART by
default
//on home computers its screen terminal module, linked to the VMEX
(getch/putch only)
//bigger VMEX/ASLIX application can use full terminal interface, still the same
way
//there will be NO spacial screen/keyboard access, only this unified terminal
way
//so, effectively this means need to develop UNIFIED TERMINAL for all HC
around...
//
//truth is, that OUR support will be probably 2 color/mono (fore/back, inverse)
+ cursor controls ONLY
//and standardized around basic ASCII subset, with probably custom loaded
FONT to be exact everywhere
//
//will be good to follow operation of SALLY basic interpreter in relation to line
terminal usage !!!
//llowing co copy/paste text outputs and to have them PASTEABLE back as
inputs for simple save/load !!!
//
//in fact, minimal monitor will serve as bootloader too (Unlock,Lock,* set
pc,Mem,Go)
//there will be built-in basic security - PIN to unlock monitor (10 tries counting
```

down, then permanent lock)

//microchip XC8 license monthly ?41

//https://www.microchip.com/en-us/development-tool/SW006021-SUB#

//Mixworx SciTE configuration for syntax colored ASLIX

//https://uloz.to/file/hFgGAJbPd2sl/mixworx-scite-

zip#lZJzjAmR2A2LjMQN4ZJD5AQtkAwt3ZxymZzEGlJlwdKukEwx4ZD==

```
#include "VMEX.h"
```

```
#ifndef ARDUINO
```

```
int main()
```

```
{
```

```
    #ifndef RUN_TESTS
```

```
    TU8 testprog[] =
```

```
    {
```

```
        0x11,0x10,0x00,0x00, //addi r1 r0 0 //reorder registers in binary encoding "l" as first  
target !!!
```

```
        0x11,0x10,0x05,0x00, //addi r1 r0 5
```

```
    };
```

```
    return vmex(testprog, sizeof(testprog));
```

```
    #else
```

```
    return runtests();
```

```
    #endif
```

```
} //main
```

```
#endif
```

```
#include "arduino.h"
```

```
#include "console.h"
```

```
#include "muldiv.h"
```

```
#define PROG_SIZE (PAGE_SIZE * 1)
```

```
#define DATA_SIZE (PAGE_SIZE * 1)
```

```
// Define core struct
```

```
typedef struct
```

```
{
```

```
    REGS_TYPE regs[16];
```

```

    TU8 rsvd[42 << 2]; //reserved (msg in/out buffers, breakpoints)
    PC_TYPE pc;
    TU8 data[DATA_SIZE];
    TU8 prog[PROG_SIZE];
} Core; //__attribute__((aligned(4))) Core;

```

```

Core core = {
    {0}, // initialize regs to all zeroes
    {0}, // initialize rsvd to all zeroes
    0, // initialize pc to zero
    {0}, // initialize data to all zeroes
    {0} // initialize prog to all zeroes
};

```

```

//main
int vmex(const TU8 testprog[], int progsz)

```



- > To create a simple monitor program for the VMEX emulator with a REPL, you can use the following template code. The monitor will display a prompt, wait for user input, and process the command. In this example, we'll support the 'r' command to display register contents.

c

Copy code

```

#include <stdio.h> #include <string.h> // Include VMEX header and other headers
// Core structure and other definitions // ... // Function prototypes void
print_registers(Core *core); void process_command(Core *core, const char
*command); int vmex(const TU8 testprog[], int progsz) { // Initialize the core
// ... // Load the testprog into the core's prog memory memcpy(core.prog,
testprog, progsz); // REPL loop char command[32]; while (1) { printf("> ");
fgets(command, sizeof(command), stdin); // Remove the newline character from the
input command[strcspn(command, "\n")] = 0; // Process the entered command
process_command(&core, command); } return 0; } void print_registers(Core *core)
{ for (int i = 0; i < 16; i++) { printf("R%d: 0x%08X\n", i, core->regs[i]); } }
void process_command(Core *core, const char *command) { if (strcmp(command, "r")
== 0) { print_registers(core); } else { printf("Unknown command: %s\n",
command); } }

```

In this template, the `vmex()` function initializes the core structure and enters the REPL loop. The `process_command()` function processes the user's input, and in this example, we only handle the 'r' command to print register contents. You can add more commands by expanding the `process_command()` function with additional conditions.

Note that this code is just a template, and you may need to adjust it to fit your specific VMEX implementation and requirements. Also, this code uses standard I/O functions like `printf()` and `fgets()`, which may not be available or suitable for all platforms. If you're

targeting an embedded system or a platform without standard I/O, you will need to replace these functions with appropriate alternatives.



well, I misguided you, as `vmex()` function is the emulator itself, place code for monitor into `monitor()` function rather



I apologize for the confusion. You can create a separate `monitor()` function to handle the interactive REPL for the VMEX emulator. Here's a revised version of the template code that separates the monitor functionality into its own function:

c

Copy code

```
#include <stdio.h> #include <string.h> // Include VMEX header and other headers
// Core structure and other definitions // ... // Function prototypes void
print_registers(Core *core); void process_command(Core *core, const char
*command); void monitor(Core *core); int vmex(const TU8 testprog[], int
progsz) { // Initialize the core // ... // Load the testprog into the core's
prog memory memcpy(core->prog, testprog, progsz); // Run the monitor REPL loop
monitor(&core); return 0; } void monitor(Core *core) { char command[32]; while
(1) { printf("> "); fgets(command, sizeof(command), stdin); // Remove the
newline character from the input command[strlen(command)] = 0; // Process
the entered command process_command(core, command); } } void
print_registers(Core *core) { for (int i = 0; i < 16; i++) { printf("R%d:
0x%08X\n", i, core->regs[i]); } } void process_command(Core *core, const char
*command) { if (strcmp(command, "r") == 0) { print_registers(core); } else {
printf("Unknown command: %s\n", command); } }
```

In this revised template, the `monitor()` function handles the interactive REPL, while the `vmex()` function initializes the core structure and calls the `monitor()` function. The rest of the code remains the same as in the previous example.

As before, this code is just a template, and you may need to adjust it to fit your specific VMEX implementation and requirements. Also, this code uses standard I/O functions like `printf()` and `fgets()`, which may not be available or suitable for all platforms. If you're targeting an embedded system or a platform without standard I/O, you will need to replace these functions with appropriate alternatives.



can you also write `main()` function from which will be by default launched the monitor prompt and in case of `/test` argument to program (main) will be executed `tests()` function?



Certainly! Here's a `main()` function that launches the monitor prompt by default and runs the `tests()` function if the `/test` argument is provided:

c

 Copy code

```
#include <stdio.h> #include <string.h> // Include VMEX header and other headers
// Core structure and other definitions // ... // Function prototypes void
print_registers(Core *core); void process_command(Core *core, const char
*command); void monitor(Core *core); int tests(); // Assuming this function is
defined elsewhere int main(int argc, char *argv[]) { // Check for "/test"
argument if (argc > 1 && strcmp(argv[1], "/test") == 0) { return tests(); } else
{ TU8 testprog[] = { 0x11, 0x10, 0x00, 0x00, // addi r1 r0 0 0x11, 0x10, 0x05,
0x00, // addi r1 r0 5 }; return vmex(testprog, sizeof(testprog)); } }
```

The **main()** function checks the command-line arguments (**argc** and **argv**) to see if **"/test"** is provided. If it is, it runs the **tests()** function. Otherwise, it initializes a test program and runs the **vmex()** function.

Make sure to include the necessary headers and definitions, and adjust the code as needed to fit your specific VMEX implementation and requirements.



where is defined C fgets



fgets is a standard C function for reading a string from a file stream. It is declared in the **<stdio.h>** header file. The function reads characters from the specified file stream into a buffer until either a newline character is encountered, the end-of-file is reached, or the specified buffer size minus one is reached.

The function prototype for **fgets** is:

c

 Copy code

```
char *fgets(char *str, int count, FILE *stream);
```

Here's a brief explanation of the parameters:

str: A pointer to the buffer that will store the read characters, including the null-terminating character.

count: The maximum number of characters to read, including the null-terminating character.

stream: A pointer to the **FILE** structure representing the file stream to read from.

fgets returns the **str** pointer on successful reads, or **NULL** if an error occurs or the end-of-file is reached before any characters are read.

Here's an example usage of **fgets**:

c

 Copy code

```
#include <stdio.h> int main(void) { char buffer[128]; printf("Enter a string:"); if (fgets(buffer, sizeof(buffer), stdin) != NULL) { printf("You entered: %s", buffer); } else { printf("Error reading input.\n"); } return 0; }
```

This example reads a string from standard input (**stdin**) using **fgets** and then prints it back to the console.



where is defined stdin?



stdin is a predefined file pointer that represents the standard input stream in C. It is typically associated with the input from the keyboard or a terminal. **stdin** is defined in the **<stdio.h>** header file.

In C, three standard I/O streams are predefined:

1. **stdin** (standard input): Used for reading input from the keyboard or a terminal.
2. **stdout** (standard output): Used for displaying output on the console or terminal.
3. **stderr** (standard error): Used for displaying error messages and diagnostics on the console or terminal.

These streams are automatically opened by the C runtime library when a program starts, so you don't need to open them explicitly. You can use them directly for reading and writing data using various I/O functions provided by the C standard library.

For example, you can use **fgets()** to read a string from **stdin**, as shown in the previous response:

c

Copy code

```
char buffer[128]; if (fgets(buffer, sizeof(buffer), stdin) != NULL) { // Process the input }
```

Similarly, you can use **fprintf()** to write formatted output to **stdout** or **stderr**:

c

Copy code

```
fprintf(stdout, "This is a message on standard output.\n"); fprintf(stderr, "This is an error message on standard error.\n");
```

However, in most cases, you can use the simpler **printf()** function to write to **stdout**:

c

Copy code

```
printf("This is a message on standard output.\n");
```



is there some alternative to fgets?



Yes, there are several alternatives to **fgets()** for reading input in C. Some of the commonly used alternatives are:

1. **fgetc()** - Reads a single character from the file stream.

c

Copy code

```
#include <stdio.h> int main(void) { int ch; printf("Enter a character: "); ch = fgetc(stdin); printf("You entered: %c\n", ch); return 0; }
```

2. **scanf()** - Reads formatted data from the standard input stream (**stdin**). You can use it to read strings, numbers, or other data types based on the format specifiers.

c

Copy code

```
#include <stdio.h> int main(void) { char buffer[128]; printf("Enter a string: "); scanf("%127s", buffer); // Note: scanf() stops reading at the first whitespace character printf("You entered: %s\n", buffer); return 0; }
```

3. **getline()** - Reads an entire line from a file stream into a dynamically allocated buffer. This function is part of the POSIX standard and may not be available on all platforms.

c

Copy code

```
#include <stdio.h> #include <stdlib.h> int main(void) { char *line = NULL; size_t len = 0; ssize_t nread; printf("Enter a string: "); nread = getline(&line, &len, stdin); if (nread != -1) { printf("You entered: %s", line); } else { printf("Error reading input.\n"); } free(line); return 0; }
```

Each of these alternatives has its own advantages and drawbacks. For instance, **fgetc()** reads only one character at a time, while **scanf()** may not handle whitespace characters as expected. **getline()** dynamically allocates memory for the input line, but it might not be available on non-POSIX platforms.

Choose the appropriate function based on your specific use case and platform requirements.





in my compiler both commented fgets and scanf are failing with compilation error

```
void monitor(Core *core)
{
    char command[32];
    while (1)
    {
        printf("> ");
        //fgets(command, sizeof(command), stdin);
        //scanf(command, sizeof(command));

        // Remove the newline character from the input
        command[strcspn(command, "\n")] = 0;

        // Process the entered command
        process_command(core, command);
    }
}
```



I apologize for the confusion. It seems I made a mistake in the usage of **scanf**. Here's the corrected version of the **monitor()** function using **scanf**:

c

Copy code

```
void monitor(Core *core) { char command[32]; while (1) { printf("> ");
scanf("%31s", command); // Use format specifier %31s to read up to 31 characters
(leave room for the null-terminating character) // Remove the newline character
from the input command[strcspn(command, "\n")] = 0; // Process the entered
command process_command(core, command); } }
```

Please note that **scanf** stops reading at the first whitespace character. If you need to read a command with spaces or other whitespace characters, you might want to use a different input method.

If you still encounter issues with both **fgets** and **scanf**, please provide the specific compilation error messages so I can help you identify and resolve the issue.



for SOME devices, both fgets and scanf give me this compilation error:

```
make -f nbproject/Makefile-default.mk SUBPROJECTS= .build-conf
make[1]: Entering directory 'W:/~prj/230326-GPT-assisted-VMEX-ISA-interpreter/VMEX_mplab/VMEX_ATSAM21G17D.X'
make -f nbproject/Makefile-default.mk dist/default/production/VMEX_ATSAM21G17D.X.production.
make[2]: Entering directory 'W:/~prj/230326-GPT-assisted-VMEX-ISA-interpreter/VMEX_mplab/VMEX_ATSAM21G17D.X'
"D:\~app\~tools\~hw-dev\microchip\xc32\v4.10\bin\xc32-gcc.exe" -g -x c -c -mprocessor=ATSAMD21G17D -fno-
common -MP -MMD -MF "build/default/production/_ext/1445274692/VMEX.o.d" -o
```

```

build/default/production/_ext/1445274692/VMEX.o ../../src/VMEX.c -DXPRJ_default=default -
mdfp="D:/~app/~tools/~hw-dev/microchip/MPLABX/v6.05/packs/Microchip/SAMD21_DFP/3.6.144/samd21d" -I
"D:/~app/~tools/~hw-dev/microchip/MPLABX/v6.05/packs/arm/CMSIS/5.4.0/CMSIS/Core/Include"
"D:/~app/~tools/~hw-dev/microchip\xc32\v4.10\bin\gcc.exe" -mprocessor=ATSAMD21G17D -o
dist/default/production/VMEX_ATSAMD21G17D.X.production.elf build/default/production/_ext/1445274692/VMEX.o
-DXPRJ_default=default -Wl,--defsym=__MPLAB_BUILD=1,-
Map="dist/default/production/VMEX_ATSAMD21G17D.X.production.map",--
memorysummary,dist/default/production/memoryfile.xml -mdfp="D:/~app/~tools/~hw-
dev/microchip/MPLABX/v6.05/packs/Microchip/SAMD21_DFP/3.6.144/samd21d"
nbproject/Makefile-default.mk:137: recipe for target 'dist/default/production/VMEX_ATSAMD21G17D.X.production.' failed
make[2]: Leaving directory 'W:/~prj/230326-GPT-assisted-VMEX-ISA-interpreter/VMEX_mplab/VMEX_ATSAMD21G17D.X'
nbproject/Makefile-default.mk:88: recipe for target '.build-conf' failed
make[1]: Leaving directory 'W:/~prj/230326-GPT-assisted-VMEX-ISA-interpreter/VMEX_mplab/VMEX_ATSAMD21G17D.X'
nbproject/Makefile-impl.mk:39: recipe for target '.build-impl' failed
d:\~app\~tools\~hw-dev\microchip\xc32\v4.10\bin\bin\..\..\lib\gcc\pic32c\8.3.1\..\..\bin\bin\pic32c-ld.exe:
d:/~app/~tools/~hw-dev/microchip\xc32\v4.10\bin\bin\..\..\lib\gcc\pic32c\8.3.1\..\..\pic32c/lib/thumb/v6-
m/nofp\libpic32c.a(read.o): in function `read':
/data01/build/bamboo/xml-data/build-dir/M32DEV-PCB4100-JOB1/build-Linux/pic32c-libs/libpic32c/stubs/read.c:63:
warning: Linking stub _mon_getc(). Hint: Provide an app-specific implementation of int _mon_getc(int canblock)
d:\~app\~tools\~hw-dev\microchip\xc32\v4.10\bin\bin\..\..\lib\gcc\pic32c\8.3.1\..\..\bin\bin\pic32c-ld.exe:
d:/~app/~tools/~hw-dev/microchip\xc32\v4.10\bin\bin\..\..\lib\gcc\pic32c\8.3.1\..\..\pic32c/lib/thumb/v6-
m/nofp\libpic32c.a(write.o): in function `write':
/data01/build/bamboo/xml-data/build-dir/M32DEV-PCB4100-JOB1/build-Linux/pic32c-libs/libpic32c/stubs/write.c:62:
warning: Linking stub _mon_putc(). Hint: Provide an app-specific implementation of void _mon_putc(char c)
d:\~app\~tools\~hw-dev\microchip\xc32\v4.10\bin\bin\..\..\lib\gcc\pic32c\8.3.1\..\..\bin\bin\pic32c-ld.exe Error: A heap
is required, but has not been specified

Info: Loading file: ./D:/~app/~tools/~hw-
dev/microchip/MPLABX/v6.05/packs/Microchip/SAMD21_DFP/3.6.144/samd21d\xc32/ATSAMD21G17D/ATSAMD21G17D.ld
collect2.exe: error: ld returned 255 exit status
make[2]: *** [dist/default/production/VMEX_ATSAMD21G17D.X.production.] Error 255
make[1]: *** [.build-conf] Error 2
make: *** [.build-impl] Error 2

BUILD FAILED (exit value 2, total time: 1s)

```



The error messages suggest that the issues you're experiencing are not directly related to **fgets** or **scanf**. The linker errors indicate that your application is missing a heap and certain functions, such as **_mon_getc** and **_mon_putc**, which are required for input/output operations.

The first step to address these issues is to add a heap to your application. You can do this by specifying a heap size in the linker script. Open the linker script file (**.ld** file) used for your project and add the following line:

arduino

Copy code

```
__heap_size = 0x1000; /* Adjust this value according to your application's
requirements */
```

Next, you need to provide implementations for the `_mon_getc` and `_mon_putc` functions. These functions are used by the standard I/O library to read and write characters. You should replace the stubs with your own implementations tailored to your target hardware. Below is a template for implementing these functions:

c

 Copy code

```
#include <stdio.h> int _mon_getc(int canblock) { // Implement reading a
character from your hardware // 'canblock' indicates if the function can block
(wait) for a character // If 'canblock' is non-zero, the function should wait
until a character is available // If 'canblock' is zero, the function should
return -1 if no character is available int ch = 0; // Replace this with the
actual character read from your hardware return ch; } void _mon_putc(char c) {
// Implement writing a character to your hardware // 'c' is the character to be
written // Add your hardware-specific code to send the character }
```

Modify the template above to provide the hardware-specific implementations for reading and writing characters on your target platform. Once you've made these changes, rebuild your project and the compilation errors should be resolved.



is possible that heap is required for stdio input functions?



Yes, it is possible that heap memory is required for standard input/output functions in C. When you use **stdio** functions such as **scanf**, **printf**, **fgets**, etc., they may rely on dynamic memory allocation for internal buffering and data processing. Dynamic memory allocation uses the heap.

If your application does not have a heap, or if the heap size is insufficient, you might encounter issues when using these functions. Therefore, it is important to ensure your application has an appropriate heap size configured, as mentioned in my previous response.

Keep in mind that the actual heap requirements will depend on the specific implementation of the C library you are using and the functions you call in your application. Some implementations might use more heap memory than others, and some might have alternatives that do not rely on the heap.



is there any possibility to use stdio or any kind of input without heap?



Yes, it is possible to use I/O functions without relying on the heap, but you'll have to implement custom I/O functions specific to your hardware instead of using the standard

stdio library. The custom functions should avoid dynamic memory allocation and use statically allocated buffers or direct hardware access to perform I/O operations.

For example, you can implement custom **getc** and **putc** functions for reading and writing characters:

c

 Copy code

```
int custom_getc(void) { // Implement reading a character from your hardware
    int ch = 0; // Replace this with the actual character read from your hardware
    return ch; }
void custom_putc(char c) { // Implement writing a character to your hardware
    // 'c' is the character to be written // Add your hardware-specific code to send the character }
```

Modify the template above to provide the hardware-specific implementations for reading and writing characters on your target platform.

After implementing these custom functions, you can use them in your **monitor()** function like this:

c

 Copy code

```
void monitor(Core *core) { char command[32]; while (1) { printf("> "); int i = 0; while (1) { int ch = custom_getc(); if (ch == '\n' || ch == '\r') { break; } if (i < sizeof(command) - 1) { command[i++] = (char)ch; } } command[i] = 0; // Terminate the command string // Process the entered command process_command(core, command); } }
```

This code reads characters using the custom **custom_getc()** function and stores them in the **command** buffer. It stops reading characters when it encounters a newline or carriage return. Note that this example assumes your custom **custom_getc()** function blocks until a character is available. If your function is non-blocking, you'll need to add a check for the "no character available" condition.

By using custom I/O functions that don't rely on dynamic memory allocation, you can avoid the need for a heap in your application. However, you might still need to adapt other parts of your code to avoid heap usage if other library functions depend on it.



is there any character input function not relying on heap in stdio?



In the standard C library (**stdio.h**), there are no functions explicitly guaranteed not to use the heap for character input. However, **getchar()** and **fgetc(stdin)** are the simplest input functions and might not use heap memory for input buffering, depending on your compiler and target platform.

It's important to note that the use of heap memory by the standard I/O functions depends on the implementation of the C library for your specific platform, and it can vary between different compilers and target hardware.

If you want to avoid heap usage completely and ensure compatibility across different platforms, it is generally recommended to implement custom input functions specific to your hardware, as described in my previous response. This way, you have full control over memory usage and can ensure that your custom functions do not use the heap.



give me example for getchar()



getchar() is a simple function that reads a single character from the standard input (usually the keyboard). Here's an example that demonstrates how to use **getchar()**:

c

Copy code

```
#include <stdio.h> int main() { int ch; printf("Please enter a character: "); ch = getchar(); printf("You entered: '%c'\n", ch); return 0; }
```

In this example, we call **getchar()** to read a character from the standard input. The function returns an integer, which is the ASCII value of the input character. We store it in the variable **ch** and then print it using the **printf()** function.



how differs sprintf and snprintf



Both **sprintf** and **snprintf** are functions used to format and store a string in a character buffer. However, there are important differences between the two:

1. **sprintf**:

It does not check the size of the destination buffer, which can lead to a buffer overflow if the output string is larger than the buffer size.

Its function signature is: **int sprintf(char *str, const char *format, ...);**

It returns the number of characters written to the buffer (excluding the null-terminator) or a negative value if an error occurs.

2. **snprintf**:

It checks the size of the destination buffer and prevents buffer overflow by limiting the number of characters written to the buffer based on the specified size.

Its function signature is: **int snprintf(char *str, size_t size, const char *format, ...);**

It returns the number of characters that would have been written if the buffer was large enough (excluding the null-terminator) or a negative value if an error occurs.

The main difference is that **snprintf** is safer than **sprintf** because it prevents buffer overflow by limiting the number of characters written based on the specified buffer size. It is generally recommended to use **snprintf** instead of **sprintf** to avoid potential security issues and undefined behavior caused by buffer overflows.

Here's a simple example illustrating the use of **snprintf**:

c

 Copy code

```
#include <stdio.h> int main() { char buffer[20]; int a = 42; float b = 3.14f;
snprintf(buffer, sizeof(buffer), "a: %d, b: %.2f", a, b); printf("Formatted
string: %s\n", buffer); return 0; }
```

In this example, the **snprintf** function formats the string and stores it in the **buffer**. The size of the buffer is passed as the second argument to prevent buffer overflow.



how to declare C function with variable number of params as printf to implement bridge?



In C, you can declare a function with a variable number of arguments using the **stdarg.h** header, which provides a set of macros to handle functions with variable arguments. Here's an example of how to implement a simple bridge function for **printf**:

c

 Copy code

```
#include <stdio.h> #include <stdarg.h> void my_printf(const char *format, ...) {
va_list args; va_start(args, format); vprintf(format, args); va_end(args); } int
main() { int a = 42; float b = 3.14f; char *s = "Hello, world!"; my_printf("a:
%d, b: %.2f, s: %s\n", a, b, s); return 0; }
```

In this example, we define a **my_printf** function with a variable number of arguments. The **va_list** type is used to hold the variable arguments. The **va_start** macro initializes **args** to retrieve the additional arguments after the **format** parameter. The **vprintf** function is used to pass the **format** string and the variable argument list **args** directly. Finally, the **va_end** macro is used to clean up the memory assigned to **args**.





ya, something like this needs to be implemented in console.h to be used through the VMEX implementation, to handle transparently full desktop console vs embedded serial console



You can create a bridge function in **console.h** that handles the transparent redirection of output to either the full desktop console or the embedded serial console, depending on your platform. Here's an example of how you can implement this:

c

Copy code

```
// console.h #ifndef CONSOLE_H #define CONSOLE_H #include <stdarg.h> void
console_printf(const char *format, ...); #endif // CONSOLE_H
```

c

Copy code

```
// console.c #include "console.h" #include <stdio.h> #ifdef EMBEDDED_PLATFORM
#include "serial.h" // Assuming you have a serial library for the embedded
platform #endif void console_printf(const char *format, ...) { va_list args;
va_start(args, format); #ifdef EMBEDDED_PLATFORM // Assuming you have a
serial_vprintf function in your serial library for the embedded platform
serial_vprintf(format, args); #else vprintf(format, args); #endif va_end(args);
}
```

In this example, you would define **EMBEDDED_PLATFORM** for your embedded platform's build system. When building for an embedded platform, the **console_printf** function will use the serial output (assuming you have a **serial_vprintf** function in your serial library). When building for a desktop platform, it will use the standard **vprintf** function.

You can then use **console_printf** throughout your VMEX implementation to handle console output in a platform-agnostic way.



exactly, thanks



You're welcome! If you have any more questions or need further assistance, feel free to ask. Good luck with your VMEX implementation and the console integration!



show me some code implementing custom fgets suing UART serial



Here's an example of how you can implement a custom **fgets**-like function using a simple UART serial library. This example assumes that you have a **uart_read** function that reads a single character from the UART. You may need to adjust the code based on the specifics of your UART library.

c

 Copy code

```
// uart_serial.h #ifndef UART_SERIAL_H #define UART_SERIAL_H #include <stddef.h>
char *uart_fgets(char *s, int size); #endif // UART_SERIAL_H
```

c

 Copy code

```
// uart_serial.c #include "uart_serial.h" #include "uart.h" // Assuming you have
a UART library with a uart_read() function char *uart_fgets(char *s, int size) {
if (size <= 0) { return NULL; } int i = 0; char c; while (i < (size - 1)) { c =
uart_read(); // Read a character from the UART if (c == '\r' || c == '\n') //
Check for end of line { break; } s[i++] = c; } // Null-terminate the string s[i]
= '\0'; // Return the string if at least one character was read, otherwise
return NULL return (i > 0) ? s : NULL; }
```

In this example, the **uart_fgets** function reads characters from the UART one by one and stores them in the provided buffer **s**. The function stops reading when it encounters a newline (**\n**) or carriage return (**\r**) character, or when the buffer is full. The resulting string is null-terminated. If at least one character was read, the function returns the buffer pointer **s**; otherwise, it returns **NULL**.



how to detect C compilation for desktop?



You can use preprocessor macros to detect the target platform during compilation. For example, you can check for specific compiler-defined macros that are set when compiling for a desktop environment.

Here's a simple example to detect if your code is being compiled for a desktop platform:

c

 Copy code

```
#if defined(_WIN32) || defined(_WIN64) || defined(__unix__) || defined(__unix)
|| (defined(__APPLE__) && defined(__MACH__)) #define DESKTOP_PLATFORM #endif
```

This code checks for common pre-defined macros for Windows, Unix, and macOS platforms. If any of these macros are defined, the code defines the custom **DESKTOP_PLATFORM** macro. You can then use this macro to conditionally compile code for a desktop platform:

```
#ifndef DESKTOP_PLATFORM // Code for the desktop platform #else // Code for other  
platforms (e.g., embedded systems) #endif
```

Keep in mind that this approach relies on the assumption that the given macros are defined by the compiler. If you're using a less-common compiler or platform, you might need to check the documentation for the appropriate pre-defined macros.
