



## VPX-1000 SERIES USER MANUAL:

### LICENSE:

MIT License

Copyright (c) 2026 mohe126



Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### INTRODUCTION:

VPX-1000 is a headless C Library that provides a VM (Virtual machine) to allow execution of VPX instructions or programs regardless of the host system, IO can be implemented freely and any system specific operations (or IO) are handled by the user of the library, which allows for customizability and control.

This in turn can make it suitable for the intended goal or task you are willing to achieve using VPX.

VPX can be compiled on any C compiler without exceptions, Regardless of the system or CPU it is compiling for. In the only case where the system provides insufficient RAM or program storage memory will it fail to compile.

Furthermore, it is not advised to use VPX for 8 or 16 bit systems unless a specifically made version of VPX was designed for these targets. As the main system is intended for 32 bit systems optimally, and allows use in 64 bit systems.



## REGISTERS:

VPX has a form of register system indexed with 8 bits, totalling 256 different registers (r0-r255). 2 of which are special purpose being r253 (RPC) and r254 (RSP) with r255 (RHC) being just a recommendation to use for hostcalls (Explained later) but is not strictly enforced.

All registers are 32 bit wide and any default 8 bit instruction operation usually ends up using the least significant byte of the register, and 16 bit being the least significant half word.

The RPC is a register that points to the current memory address being indexed, behaving like the program counter in most real life counterparts when it comes to CPUs.

The RSP register is the stack pointer, pointing to the stack which is accessed or used with PUSH, POP, CALL and RET instructions. The stack pointer is equal to 0 by default upon the first instance of VM execution. Afterwards it will retain its value. Thus initializing it to a specific memory address (segmented) is recommended.

## C IMPLEMENTATION:

```
uint32_t vpx1000_cpu_registers[256] = {  
    0,  
    //r253: RPC.  
    //r254: RSP.  
    //r255: RHC.  
};
```

## MEMORY:

Memory is aligned in a big-endian format, and that was chosen for the sake of simplicity when writing bare machine code, though that can make memory access slower so if required it is possible to edit the code to account for small-endian format which is usually the case for the host CPU, saving instructions and allowing for faster execution.

The memory range is of the 32 bit's integer limit range, allowing for a maximum of 4 GB of memory for use, but usually the actual use case requires much less.

All memory operations have 8, 16, and 32 versions.

Memory has different operation modes, SAFE and UNSAFE, during initialization of VPX, it is possible to select the mode which will use different function pointers depending on the mode, the functions responsible for the SAFE operations have bounds checking, which can ensure execution safety at the cost of performance, whereas UNSAFE allows for arbitrary memory access with no bounds checking whatsoever. Which can improve performance if used correctly, but is usually not recommended.

## C IMPLEMENTATION:

```
//[[ MEMORY ]]  
uint8_t* vpx1000_mem_ptr = vpx1000_NULL;  
uint32_t vpx1000_mem_size = 0;
```

## SAFE FUNCTIONS:

```
uint8_t vpx1000_mem_sf_read8(uint32_t adr) {  
    if(adr >= vpx1000_mem_size){  
        vpx1000_f_error_code = 1; //Memory read error.  
        vpx1000_f_error_value = adr;  
        return 0;  
    }  
    return vpx1000_mem_ptr[adr];  
  
}  
uint16_t vpx1000_mem_sf_read16(uint32_t adr){  
    if(adr + 1 >= vpx1000_mem_size){  
        vpx1000_f_error_code = 1; //Memory read error.  
        vpx1000_f_error_value = adr;  
        return 0;  
    }  
    return (vpx1000_mem_ptr[adr] << 8) | vpx1000_mem_ptr[adr+1];  
}  
uint32_t vpx1000_mem_sf_read32(uint32_t adr){  
    if(adr + 3 >= vpx1000_mem_size){  
        vpx1000_f_error_code = 1; //Memory read error.  
        vpx1000_f_error_value = adr;  
        return 0;  
    }  
    return (vpx1000_mem_ptr[adr] << 24) | (vpx1000_mem_ptr[adr+1] << 16) |  
(vpx1000_mem_ptr[adr+2] << 8) | vpx1000_mem_ptr[adr+3];  
}
```

## UNSAFE FUNCTIONS:

```
uint8_t vpx1000_mem_uf_read8(uint32_t adr) {
    //No memory bounds check.
    return vpx1000_mem_ptr[adr];
}

uint16_t vpx1000_mem_uf_read16(uint32_t adr) {
    return (vpx1000_mem_ptr[adr] << 8) | vpx1000_mem_ptr[adr+1];
}

uint32_t vpx1000_mem_uf_read32(uint32_t adr) {
    return (vpx1000_mem_ptr[adr] << 24) | (vpx1000_mem_ptr[adr+1] << 16) | (vpx1000_mem_ptr[adr+2] << 8) | vpx1000_mem_ptr[adr+3];
}
```

For further information on how read, write, fetch, pop or push operations are implemented, please check the source code for VPX-1000 or any version you are interested in.

## ERROR HANDLING:

If any instruction fails in any case, or a memory access violation (Safe mode) happens. VPX will log an error which will exit the primary execution loop in the form of a special case hostcall (More on that later) and log error codes to the variables:

```
uint8_t vpx1000_f_error_code = 0;
uint32_t vpx1000_f_error_value = 0;
```

In the example of an invalid memory access, the error code will encode a value of the type of error, 1 for memory read errors for example as you can see:

```
uint8_t vpx1000_mem_sf_read8(uint32_t adr) {
    if(adr >= vpx1000_mem_size) {
        vpx1000_f_error_code = 1; //Memory read error.
        vpx1000_f_error_value = adr;
        return 0;
    }
    return vpx1000_mem_ptr[adr];
}
```

The error value instead is the value that caused the error, in this case the address that the read operation attempted to access.

This table contains all error codes and their meaning:

INVALID MEMORY READ ERROR	1
INVALID MEMORY WRITE ERROR	2
INVALID MEMORY FETCH ERROR:	3
INVALID MEMORY PUSH ERROR:	4
INVALID MEMORY POP ERROR:	5
INVALID OPCODE ERROR:	255*

NOTES: An invalid OPCODE error will contain the opcode itself in the error value. However, in the unsafe mode it will treat any invalid OPCODE as a NOP instruction.

## BASIC USE:

This example will show how to use VPX-1000, this example is extremely simple and will not include everything however it is a good starting point for explanation. The code will be explained later in detail.

```
///[[ EXAMPLE ]]
#include <stdio.h>
#include <stdint.h>

#include "VPX-1000/vpx-1000.h"

//Optional macros
#define SAFE 1
#define UNSAFE 0

int main(){
    uint8_t program[256] = {
        130, 0, 0, 0, 0, 0, //MOVI, r0, 0x0 ;Will contain jump index.
        130, 1, 0, 0, 0, 0, //MOVI, r1, 0x0 ;Will contain increment counter.
        130, 2, 0, 0, 0, 255, //MOVI, r2, 0xFF ;Will contain decrement counter.
        //[[ NOTE: by default all these registers should be 0, but this code. ]]
        //[[ Assumes otherwise for showcasing good practice. ]]
        38, 2, //DEC, r2 ;Decrement r2 for the loop.
        37, 1, //INC, r1 ;Increment r1 to see counter results
        22, 2, //CMPZ, r2 ;Compare r2 with 0.
        148, 0xFF, 0xFF, 0xF9, //JMPFI, -7 ;Jump to -7 in two's complement if false.
        //[[ OTHERWISE: ]]
        1,    //HostCall ;Exit program and return to host
    };
    uint32_t size = 256;

    vpx1000_init(program, size, SAFE);

    vpx1000_start();

    printf("Host Called.\n");
    printf("Current r1 Value:%u\n", vpx1000_cpu_rreg(1));
    printf("Current rpc Value:%u\n", vpx1000_cpu_rreg(vpx1000_RPC));

    return 0;
}
```

```
Now compile with: "gcc main.c VPX-1000/vpx-1000.c -o vpx-test.exe"
```

Output should look something like:

```
=====
```

```
Host Called.
```

```
Current r1 Value:255
```

```
Current rpc Value:30
```

```
=====
```

Now the details of how everything works:

```
#include <stdio.h>
#include <stdint.h>

#include "VPX-1000/vpx-1000.h"
```

Includes standard C libraries and vpx-1000.h, which should be inside a folder called “VPX-1000”.

```
//Optional macros
#define SAFE 1
#define UNSAFE 0
```

Optional macros for the values for safe and unsafe mode.

```
int main() {
    uint8_t program[256] = {
        130, 0, 0, 0, 0, 0, //MOVI, r0, 0x0 ;Will contain jump index.
        130, 1, 0, 0, 0, 0, //MOVI, r1, 0x0 ;Will contain increment counter.
        130, 2, 0, 0, 0, 255, //MOVI, r2, 0xFF ;Will contain decrement counter.
        //[[ NOTE: by default all these registers should be 0, but this code. ]]
        //[[ Assumes otherwise for showcasing good practice. ]]
        38, 2, //DEC, r2 ;Decrement r2 for the loop.
        37, 1, //INC, r1 ;Increment r1 to see counter results
        22, 2, //CMPZ, r2 ;Compare r2 with 0.
        148, 0xFF, 0xFF, 0xFF, 0xF9, //JMPFI, -7 ;Jump to -7 in two's complement if
false.
        //[[ OTHERWISE: ]]
        1, //HostCall ;Exit program and return to host

    };
}
```

Array that contains a test program which counts down from 255 to 0 then exits.

```
1, //HostCall ;Exit program and return to host
```

Host calls are instructions used to exit from the vpx1000\_start(); execution loop

They essentially work like system calls or interrupts but for the host.

vpx1000\_start(); will return 0 if the exit was caused by a hostcall, Otherwise a 1 if it was due to an error.

```
uint32_t size = 256;
```

Variable to contain the size of program memory, which allows the VM to know where the memory bounds are set, Maximum value is the 32 bit integer limit.

```
vpx1000_init(program, size, SAFE);
```

Initializes vpx1000. This step is required before starting the execution.

```
vpx1000_start();
```

Starts VM, will continue executing until an error is met or a hostcall instruction was called.

```
printf("Host Called.\n");
printf("Current r1 Value:%u\n", vpx1000_cpu_rreg(1));
printf("Current rpc Value:%u\n", vpx1000_cpu_rreg(vpx1000_RPC));
```

Log information to make sure the VM worked.

## INSTRUCTION SET ARCHITECTURE:

The instruction set for VPX-1000 has 91 instructions, 55 of them being regular instructions and 36 of them being immediate variants.

This table summarizes all instructions and their opcodes, for details on all the instructions, operands and how they work, please refer to the ISA manual.

## NOTES AND INFORMATION:

This is the first version of the user manual intended to be as compact as possible. And it may include errors, inaccuracies or missing information.

If you find any issues please make your own version of the manual or fix issues and notify me. All help is greatly appreciated.

