Security Aspects in Software Development KU 2013/2014 "TinyVM (C2)"

Daniel Hein Johannes Winter
Thomas Kastner
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, 8020 Graz, Austria
`sicherheitsaspekte@iaik.tugraz.at`

December 19, 2013

Contents

1 Introduction

Assignment title:	TinyVM (C2)
Group size:	2 students
Start date:	December 19, 2013
Maximum score:	35 points (35% percent of final* mark)
HARD SUBMISSION DEADLINE:	**January 20, 2014 (23:59:59 CET)**

See Section 3 for details on the submission process.
See Section 4 for the subtasks and questions of this assignment.
* Bar the oral exam.

1.1 Motivation

In this task we focus on secure implementation of a small virtual machine, and the associated assembler, in the C programming language. As part of implementing this assignment, you will have to deal with text data and binary data in different flavours. Moreover you will be confronted with highly dynamic and potentially untrusted user input[1].

According to the *CWE/Sans Top 25 Most Dangerous Software Errors* list[1], buffer overflows and variants of integer overflows are still amongst the top 25 software security problems. Classic buffer overflows even rank at place 3 of this list (immediately after SQL and OS command injections).

1.2 Objectives

The goal of this exercise is to improve your skills in secure C programming. In particular this exercise is intended to train your ability:

- to securely handle text and binary data in memory unsafe languages like C;
- to write robust code that can tolerate invalid and out of bounds inputs from potentially untrusted external sources; As part of this exercise, you should improve your skills in defensive programming.

1.3 Plagiarism

⚠ *Plagiarism will not be tolerated and any contributions containing copied code will automatically receive a negative rating. It is explicitly allowed to use material and code snippets shown during the lecture or the exercises.*

2 Software requirements and setup

The reference platform for this exercise is the VMware[2] image, which can be found in the download section [2] of the course website. The reference image is based on Xubuntu(Precise Pangolin 12.04 LTS) Linux with Apache Tomcat 7.0.12, Oracle JAVA SDK 1.7.0_40, MySQL 5.5.32, lemon 3.7.9, gperf 3.0.3, clang 3.0.6 and CUnit 2.1.2(unstable) and is shipped with a pre-installed Eclipse(Kepler).

[1] In the end the input to your virtual machine is a small (bytecode) program

[2] `http://www.iaik.tugraz.at/content/teaching/master_courses/sicherheitsaspekte_in_der_softwareentwicklung/practicals/downloads/`

⚠ *If you prefer to use your own setup, bear in mind that we can and will only support the reference platform virtual appliance. We can give no guarantees about the compatibility of the software used in this assignment with non-reference platform setups.*

In the remainder of this document we assume that you are using our VMware reference image. Unless explicitly noted otherwise, all command line examples shown here are to be run inside the virtual environment of the reference image and *not on your host computer.*

2.1 Updating your repository

As for the previous exercises, the source code for this assignment is distributed as a Git patch to be applied to your repositories. To integrate the updates for this task into your repository you simply have to download the patch-file from `https://bigfiles.iaik.tugraz.at/get/2af2ff3e9b093d55e71150794fa6d08b` and merge it into your local working copy.

To merge the changes for this assignment run the following commands in your working directory:

```
# Download the patch-file for task C1 into your home-directory
sase@vm:~$ wget -O ~/assignment-c2.patch \
  https://bigfiles.iaik.tugraz.at/get/2af2ff3e9b093d55e71150794fa6d08b

# Switch to your local working copy.
sase@vm:~$ cd ~/<your_sase_repo>

# Ensure that you are on the master branch of your repository
# (We don't want to accidentially clobber a submission branch of
#  a previous task)
sase@vm:~/<your_sase_repo>$ git checkout master

# Merge the patch into your working directory using "git am".
sase@vm:~/<your_sase_repo>$ git am ~/assignment-c2.patch
```

2.2 External libraries

For this assignment your submission *MUST NOT* use any external C libraries, except for the standard C library (libc) and the preinstalled CUnit library (for the unit testing code). The build system provided with the assignment patch is already preconfigured for these libraries.

It must be possible to successfully compile and run your programs on the VMware reference image! For details on how you can find out if your submission compiles on our submission system, see Section 3.3.

2.3 CUnit

To help you with testing your implementation, we added a set of CUnit tests, which cover basic functionality checks on the functions you have to implement as part of the tasks in Section 4.

⚠ *Please be aware that passing all of those checks does not automatically mean that your implementation is fully correct or secure. The tests only provide minimal functional checks for this task.*

3 Submission

The submission for this tasks consists of all source files, which are required to compile your solutions to the tasks discussed in Section 4. Your solutions *MUST* be compilable by our test system without requiring any modifications to the test system. See Section 3.3 on how you can get intermediate feedback about the build status of your submission.

Your submission should at least contain:

- All source files required to build your submission. They *MUST* be compilable with the build infrastructure from the assignment source patch. See the checklist in Section 3.5 for details.

3.1 Creating the submission branch

The submission itself is done via a separate `submission-c2` branch in your Git repository. To create this branch on the server, you need to execute the following steps *once*:

```
sase@vm:~/<your sase repo>$ git checkout -b submission-c2
sase@vm:~/<your sase repo>$ git push -u origin submission-c2
```

The “checkout” command (with `-b` option) first creates a new local branch in your local repository. The “push” command then publishes your local branch on the server. The `-u` option to the “push” command tells Git to setup tracking for your new branch.

Your group partner needs to setup a local copy of the submission branch as well. To do so, run the following steps in your group partner’s local Git repository *once* after you have pushed the submission branch to the server:

```
sase@vm:~/<partner's sase repo>$ git pull
sase@vm:~/<partner's sase repo>$ git checkout -t -b submission-c1 remotes/origin/submission-c2
```

The “pull” command in your colleague’s repository fetches changes from the server. The “checkout” command creates a local “submission-c2” branch (`-b` option), which is configured to track (`-t` option) the “submission-c2” branch on the server.

You can use the normal `git push` and `git pull` commands, after you and your colleague have finished the one-time setup of the submission branch.

☞ *We strongly recommend to read `http://git-scm.com/book/en/Git-Branching-Remote-Branches` if you have not worked with Git (remote) branches before.*

⚠ *We recommend, to actively use the Git repository, and to regularly merge and push your current development progress into the `submission-c2` branch. (You can even use the submission branch as main development branch.)*

This way, it becomes much less likely to miss any important changes or files for your final submission.

3.2 Working with the submission branch

Once you and your colleague have configured the submission branch, you can work with it like any other Git branch. You can directly commit to the submission branch via “git commit” or merge changes from (local) working branches via “git merge”.

3.3 Automated compilation

For this task, we will use an automated build system, to regularly compile your submissions in our reference environment. Usually the builds are scheduled at 4:00 AM every day.

Our build bot tries to checkout your submission branch using the steps discussed in section 3.4. On success, the bot then tries to build your submission using our reference build environment. The commit considered by the build bot will be tagged with a Git tag named `iaik/buildbot/c2/XX` where `XX` increases with each scheduled build.

You can fetch the tags generated by the build-bot using either `git pull` or `git fetch --tags`. The build history of your submission branch is accessible via the `git tag -l` command:

```
sase@vm:~/<your sase repo>$ git fetch --tags
sase@vm:~/<your sase repo>$ git tag -n3 -l iaik/buildbot/c2/*
 iaik/buildbot/c2/1 This commit was built successfully by IAIK's Build-Bot
   Build-Schedule: Wednesday, December 18, 2013 2:50:41 PM CET
```

```
    Build-Status: SUCCESS
  ...
```

3.4 Verifying your submission

You can easily verify your submission by cloning a fresh copy of your repository into a new directory, and trying to check out the `submission-c2` branch. The Git commands used by our build system to clone your submissions are semantically equivalent to:

```
$ git clone git@teaching.student.iaik.tugraz.at:sase2013gXX.git
$ cd sase2013gXX
$ git checkout -b submission-c2 origin/submission-c2
```

Our build-system automatically tags the commit considered as final submission, with an `iaik/submission/c2` tag, once the submission deadline has passed. We will not consider any commits after the `iaik/submission/c2` tag as submission for this task.

3.5 Checklist

Here is a checklist with all steps required to successfully complete this task:

- ☐ Download the source-patch for this task from the IAIK website and apply it to your Git repository as discussed in section 2.1.
- ☐ Create and setup your submission branch as discussed in section 3.1.
- ☐ Solve the tasks discussed in section 4 and commit the code of your solutions to your Git repository.
- ☐ Checkout your local `submission-c2` branch and ensure, that everything you want to submit is merged. Ensure that your code can be compiled with the build infrastructure from the assignment source patch.
- ☐ Push your `submission-c2` branch to our servers. You can push the submission branch as often as you like. Any changes after the submission deadline will be ignored.
- ☐ Verify that your submission builds, using the steps discussed in Section 3.3. Follow the steps in Section 3.4 to verify, that you pushed the correct code to the server.

3.6 Deadline

HARD SUBMISSION DEADLINE: January 20, 2014 (23:59:59 (CET))

⚠ *We highly recommend to submit your solution at least one or two days before the ultimate submission deadline. Last minute submissions are always risky with respect to unforeseen technical difficulties.*

4 Tasks

In the practical part of the TinyVM assignment, you are asked to implement parts of the *tinyvm* virtual machine and the supporting assembler.

☞ *The unit tests provided as part of the assignment source patch define the minimal functional requirements for the* tinyvm *assembler and the* tinyvm *virtual machine.*

4.1 *(15 points)* Byte buffers and endianess

Operating with blobs of binary data can be error prone and cumbersome task using the C programming language. To simplify handling of binary data, we define two memory abstraction: *byte buffers* and *memory views*.

In this task, you are asked, to secure operatr

Questions:

(1.a) ☐ *(8 points) Basic* byte buffers. Operating with blob of binary data often requires you to store them in some kind of dynamically grow-able buffer data structure. For example, when you are constructing a larger blob by repeatedly append small chunks of data. Both, the *tinyvm* virtual machine and the *tinyvm* assembler need this kind of growing byte buffer.

In this subtask, you have to implement the `BufferCreate`, `BufferFree`, `BufferAppend`, `BufferSkip`, `BufferClear`, `BufferGetByte`, `BufferSetByte`, `BufferGetBytes`, and `ByteBufferGetLength` functions discussed in Section A.1 to provide a basic byte buffer implementation.

(1.b) ☐ *(4 points) Accessing parts of byte buffers (*memory view*).* Operations on existing (binary) data blobs typically involves reading and writing of data-words at given offsets within the byte blobs.

Error while computing the sizes and the offset for these read and write accesses are a prime source for buffer overflows. The *tinyvm* virtual machine needs to access byte blobs, often in situations where the offsets and lengths directly come from (untrusted) bytecode programs.

In this subtask, you are asked to securely program the `MemInit`, `MemGetLength` and `MemGetBytes` functions, which are discussed in Section A.2 to provide a slightly safer (bounds checked) way to access the content of raw byte buffers.

(1.c) ☐ *(3 points) Dealing with different byte-orders* When exchanging binary data between different platforms byte-ordering (endianess) is often an serious issue. Some platforms use little-endian byte-order (least significant byte first) others use big-endian byte-orders (most significant byte first).

In this subtask, you have to implement the `MemGetHalf`, `MemSetHalf`, `MemGetWord` and `MemSetWord` functions discussed in Section A.2. These functions allow the *tinyvm* virtual machine to work with both, big-endian and little-endian data in a uniform manner. Be sure to implement proper bounds checking for your functions.

4.2 *(10 points)* Strings and parsing

Parsing of textual input data is another major source of security-critical errors in typical C programs. The spectrum of errors includes missing or bad string termination, inappropriate length computations, and unexpected side effects when using certain standard string functions with overlapping memory regions.

In this task, you are asked to securely implement parts of the string tokenizer and the lexical scanner of a the *tinyvm* assembler. Most parts of the assembler are already finished, and can be used without any need to modify them.

Once you have the token management and the lexical scanner in place, it should be very easy to produce bytecode programs for execution by the *tinyvm* virtual machine.

The *tinyvm* assembler found in the `tasks/c2/assembler` directory of the source patch implements parsing of (human) readable source code in a two steps:

In the first step, a *lexical scanner* (tokenizer) scans of the input text and splits it into simple *tokens*. These *tokens* can typically be integers, variable names, function names, keywords or certain punctuation characters. Whitespace characters and comments are discarded during the lexical analysis step.

In the next step, a parser consumes the tokens and tries to process them according to the syntax (grammar) of the *tinyvm* assembler. The parser triggers calls specific actions, whenever a syntactically valid part of input has been processed, or when a syntax error occurred.

☞ *In this task you only have to implement the* token management code *and the (simple) lexical scanner (tokenizer). You do NOT need to implement the syntactical parser; it is auto-generated from the grammar specification in `tasks/c2/assembler/grammar.y` using the Lemon parser generator tool.*

Questions:

(2.a) ☐ *(3 points) Token handling.* Implement the missing parts of the token management functions discussed in Section A.3.

⚠ *Pay particular attention to proper* `'\0'` *termination of C strings. (When passed a valid token object, the* `AsmTokenGetText` *function always returns a properly terminated string).*

(2.b) ☐ *(7 points) Lexical scanner (tokenizer).* Implement the missing parts of the lexical scanning code discussed in Section A.4. The key function of the lexical scanner is the `AsmLexerScan` function, which scans the next token from the input stream.

Have a look at the documentation (and the todos) in `tasks/c2/assembler/asm_lexer.c`, together with the provided assembler test cases it should give you a fairly good idea on how the lexical tokens look like.

☞ *The byte buffer from above may be very useful when collecting the textual content of the tokens produced by the lexical scanner.*

4.3 *(10 points)* Completing the *tinyvm* virtual machine

The

Questions:

(3.a) ☐ *(3 points) Operand stack* The *tinyvm* virtual machine needs an operand stack to execute bytecode programs. A limit for the maximum size of the operand stack is specified when the virtual machine is created.

In this subtask you are required to securely implement the `VmStackPush`, `VmStackPop`, `VmStackGetUsage` and `VmStackAt` discussed in section A.5. Note that the existing parts of the *tinyvm* virtual machine assume, that your stack management code performs proper parameter checks. In addition to the stack functions, you will also have to adapt the `VmCreate` and `VmDelete` functions, as well as the `VmContext` data structure.

(3.b) ☐ *(2 points) Object management* The object management of the *tinyvm* virtual machine are incomplete. In this subtask you will complete the missing parts of the object handling functions.

The missing object management functions are `VmCreateObject`, `VmAccessObject` and `VmGetObject` discussed in section A.5. In addition to completing this functions, you will also have to adapt the `VmCreate` and `VmDelete` functions, as well as the `VmContext` and `VmObject` data structures.

☞ *You need a data structure to manage the objects known by the virtual machine. Linked lists or arrays (of pointers) are some of the possible choices.*[3] ☞ *Use the byte buffers (from the previous tasks) to manage the payload data of your objects, to simplify bounds checking.*

(3.c) ☐ *(3 points) Bytecode loading* Implement the missing parts of the bytecode loader, to allow the virtual machine to load bytecode programs from external sources. To complete this subtask you need to implement `VmLoadByteCode` function discussed in section A.5.

Details on the bytecode format can be found in `vm/bytecodes.h`. The Doxygen documentation of extracted from `vm/bytecodes.h` is also part of the appendix of this document.

☞ *By using the memory views (from the previous tasks) you can greatly simplify the implementation of your bytecode loader. (In particular with respect to boundary checks and endianess handling).*

(3.d) ☐ *(2 points) Improving bytecode security* The current implementation of the *tinyvm* `LDW`, `STW`, `LDB`, `STB`, `UDIV`, `SDIV`, and `JMP` bytecode instructions contain implementation bugs, which can affect the security of the virtual machine.

In this subtask you should fix the problems of the given bytecode handler. The problematic handlers in `vm_arith.c`, `vm_ldst.c` and `vm_call.c` are annotated with `todo` markers.

First write *tinyvm* programs to trigger the annotated issues, and then fix them in the *tinyvm* source code. Use your “testcase” programs, to show that the fix works.

[3] For this exercise it is perfectly acceptable if your `VmGetObject` needs linear time ($O(n)$) to find an object by its handle.

A Module Documentation

A.1 Simple memory buffers

Typedefs

- typedef struct **Buffer Buffer**

  A simple byte buffer.

Functions

- **Buffer** ∗ **BufferCreate** (void)

  Creates a new empty byte buffer.
- void **BufferFree** (**Buffer** ∗buffer)

  Frees a byte buffer and releases all associated resources.
- bool **BufferAppend** (**Buffer** ∗buffer, const void ∗data, size_t len)

  Appends data to a byte buffer.
- bool **BufferSkip** (**Buffer** ∗buffer, unsigned char filler, size_t len)

  Skips bytes at the end of a buffer.
- bool **BufferClear** (**Buffer** ∗buffer)

  Discards the content of a byte buffer.
- unsigned char ∗ **BufferGetBytes** (**Buffer** ∗buffer, size_t offset, size_t len)

  Gets a pointer into the data region of the byte buffer.
- size_t **BufferGetLength** (const **Buffer** ∗buffer)

  Gets the length of a byte buffer.

A.1.1 Typedef Documentation

A.1.1.1 typedef struct Buffer Buffer

Byte buffers provide a simple abstraction for handling raw binary data. The only size limitations of a byte buffer should be the available amount of memory and the intrinsic limits of data-types such as `size_t`.

A byte buffer may contains arbitrary binary data, so it is necessary to store the length and the base pointer in the byte buffer structure.

A.1.2 Function Documentation

A.1.2.1 bool BufferAppend (Buffer ∗ *buffer,* const void ∗ *data,* size_t *len*)

This functions appends data at the end of a byte buffer. The data to be appended is specified by the `data` and `len` parameters.

Warning

The `data` and `len` parameters may refer to a slice of the byte buffer itself. Be careful when growing the buffer! (in particular if you are planning to use realloc). The following code fragment illustrates an example of this situation:

```
...
ByteBuffer *buffer = ...;

// Get a slice of the buffer
size_t len = BufferGetLength(buffer);
void *data = BufferGetBytes(buffer, 0, len);

// Append the buffer to itself
BufferAppend(buffer, data, len);
...
```

Parameters

`in,out`	*buffer*	The destination byte buffer.
`in`	*data*	Data to be appended at the end of the buffer.
	len	Length (in bytes) of the data block to be appended.

Returns

`true` if the append operation succeeded.
`false` if the append operation failed, for example due to bad arguments or an out of memory condition.

Warning

Calling this function (may) invalidates all pointers which have been returned by preceding **BufferGetBytes** (p. 15) calls.

A.1.2.2 bool BufferClear (Buffer ∗ *buffer*)

This functions discards the current content of a byte buffer and truncates the length of the buffer to zero.

Parameters

`in,out`	*buffer*	The destination byte buffer.

Returns

`true` if the append operation succeeded.
`false` if the append operation failed, for example due to bad arguments or an out of memory condition.

Warning

Calling this function (may) invalidates all pointers which have been returned by preceding **BufferGetBytes** (p. 15) calls.

A.1.2.3 Buffer∗ BufferCreate (void)

This function allocated and initializes a new empty byte buffer object.

Returns

A new empty byte buffer or NULL on error.

A.1.2.4 void BufferFree (Buffer ∗ *buffer*)

This function deletes a given byte buffer object and releases all associateds (memory) resources.

Parameters

`in`	*buffer*	The byte buffer to be deleted.

A.1.2.5 unsigned char* **BufferGetBytes** (**Buffer** * *buffer,* size_t *offset,* size_t *len*)

This function returns a pointer to a slice of the data region of the given byte buffer. The caller can use the returned pointer to directly access parts of the byte buffer for reading and writing.

This function fails with a NULL result, if the caller tries to access a region (given by `offset` and `len`) which lies outside the current bounds of the byte buffer.

The pointer returned by this function remains valid until either the buffer is freed (via **BufferFree** (p. 14)) or until the size of the buffer is changed, in response to a **BufferAppend** (p. 13), **BufferSkip** (p. 15) or - **BufferClear** (p. 14) call.

To get a raw pointer to the content of the full byte buffer the following code fragment can be used:

```
...
Buffer* buffer = ...;
size_t len = BufferGetLength(buffer);
unsigned char* data = BufferGetBytes(buffer, 0, len);
...
```

Parameters

`in,out`	*buffer*	The byte buffer to access.
`in`	*offset*	The offset of the first byte to be accessed in the buffer. (An offset of zero refers to the beginning of the buffer).
`in`	*len*	The number of bytes to be accessed. The slice returned by this function contains `len` bytes starting at `offset`.

Returns

On success, a valid pointer to the first byte of the requested part of the byte buffer.
NULL if an emtpy slice (`len` == 0) is requested.
NULL on error, for example if `buffer` is NULL or if the function is called with `offset` and/or `size` referring to a region outside the bounds of the given byte buffer.

A.1.2.6 size_t **BufferGetLength** (const **Buffer** * *buffer*)

Parameters

`in`	*buffer*	The byte buffer to be queried.

Returns

The length of the byte buffer,

A.1.2.7 bool **BufferSkip** (**Buffer** * *buffer,* unsigned char *filler,* size_t *len*)

This functions "skips" a number of bytes at the end of a buffer, by appending the `filler` value `len` times.

Parameters

`in,out`	*buffer*	is the destination byte buffer.
`in`	*filler*	is the filler value for the buffer.
	len	Length (in bytes) of the data block to be appended.

Returns

`true` if the append operation succeeded.
`false` if the append operation failed, for example due to bad arguments or an out of memory condition.

Warning

Calling this function (may) invalidates all pointers which have been returned by preceding **Buffer-GetBytes** (p. 15) calls.

A.2 Memory views

Data Structures

- struct **MemView**

  View of a region of memory.

Enumerations

- enum **MemViewFlags** { **MEM_VIEW_NORMAL** = 0, **MEM_VIEW_READONLY** = 1, **MEM_VIEW_-BIGENDIAN** = 2, **MEM_VIEW_RESERVED** = ~(MEM_VIEW_READONLY | MEM_VIEW_BIGEND-IAN) }

  Flags for memory views.

Functions

- bool **MemInit** (**MemView** ∗view, **MemViewFlags** flags, **Buffer** ∗buffer, size_t base, size_t length)

  Initializes a memory view for use.
- size_t **MemGetLength** (const **MemView** ∗view)

  Gets the length of a memory view in bytes.
- bool **MemIsReadOnly** (const **MemView** ∗view)

  Tests if a memory view is marked as read only.
- bool **MemIsBigEndian** (const **MemView** ∗view)

  Tests if a memory view uses big-endian byte-order.
- bool **MemSetBigEndian** (**MemView** ∗view, bool enable)

  Enables or disables big-endian byte-order of a memory view.
- unsigned char ∗ **MemGetBytes** (**MemView** ∗view, size_t offset, size_t length)

  Gets a raw pointer to a range of bytes in a memory view.
- bool **MemGetByte** (uint8_t ∗z, **MemView** ∗view, size_t offset)

  Reads a single byte (unsigned 8-bit integer) from a memory view.
- bool **MemSetByte** (**MemView** ∗view, size_t offset, uint8_t value)

  Writes a single byte (unsigned 8-bit integer) to a memory view.
- bool **MemGetHalf** (uint16_t ∗z, **MemView** ∗view, size_t offset)

  Reads a single unsigned half-word (16-bit integer) from a memory view.
- bool **MemSetHalf** (**MemView** ∗view, size_t offset, uint16_t value)

  Writes a single unsigned half-word (16-bit integer) to a memory view.
- bool **MemGetWord** (uint32_t ∗z, **MemView** ∗view, size_t offset)

  Reads a single unsigned word (32-bit integer) from a memory view.
- bool **MemSetWord** (**MemView** ∗view, size_t offset, uint32_t value)

  Writes a single unsigned word (32-bit integer) to a memory view.

A.2.1 Enumeration Type Documentation

A.2.1.1 enum MemViewFlags

The values of this enumeration are bit-masks used as flags of memory view objects. Each memory view can have one or more of the flags given here.

Enumerator:

MEM_VIEW_NORMAL Indicates a normal, writable memory view. This flag value indicates a normal writable memory view. The endianess depends on wether the **MEM_VIEW_BIGENDIAN** (p. 18) flag is set or not.

MEM_VIEW_READONLY Indicates a read-only memory view. This flag is mutually exclusive with the **MEM_VIEW_NORMAL** (p. 18) flag, and indicates a read-only memory view. All attempts to write to the memory view will fail.

MEM_VIEW_BIGENDIAN Indicates a big-endian memory view. This flag indicates that a memory view uses big-endian byte (instead of default little endian) byte order to store multi- byte values.

MEM_VIEW_RESERVED Bit-mask with the reserved flags. This special value contains a bit-mask with all currently reserved flags of the memory view API.

A.2.2 Function Documentation

A.2.2.1 bool MemGetByte (uint8_t ∗ *z,* MemView ∗ *view,* size_t *offset*)

This functions reads a single byte at offset `offset` from the memory view given by `view` and stores the result in `z`.

Parameters

`out`	*z*	Pointer to the variable receiving the result byte
`in`	*view*	The memory view to be accessed.
`in`	*offset*	The (byte) offset of the half-word to be accessed (relative to the beginning of the view; offset zero referes to the first byte in the view.)

Returns

`true` if the read operation was successful.
`false` if the read operation failed, for examnple due to an out of bounds access or other invalid parameters. The variable pointed to by `z` must not be modified in case of an error.

A.2.2.2 unsigned char∗ MemGetBytes (MemView ∗ *view,* size_t *offset,* size_t *length*)

This functions returns a raw pointer to a range of bytes visible in a memory view. The offset accepted by this function is relative to the beginning of the memory view. (The base offset of the memory is automatically added when calling **BufferGetBytes** (p. 15) of the underlying byte buffer.)

Note

This function does not automatically check the **MemIsReadOnly** (p. 20) flag. It is the API user's responsibility to perform such a check if desired.

Parameters

`in`	*view*	The memory view to be accessed.
	offset	Offset of the first byte (relative to the start if the memory view).
	length	Length of the byte range to be accessed.

Returns

A pointer to the byte range in the underlying byte buffer on success. See the remarks on **BufferGetBytes** (p. 15) for details on the validity of the returned pointer. (e.g. calling **BufferAppend** (p. 13) on the underlying buffer will invalidate the returned pointer).

A.2.2.3 bool MemGetHalf (uint16_t ∗ *z*, MemView ∗ *view*, size_t *offset*)

This functions reads a single unsigned half-word at offset `offset` from the memory view given by `view` and stores the result in `z`. This function honors the **MEM_VIEW_BIGENDIAN** (p. 18) flag of the memory view, to determine wether the half-word is stored in big-endian byte order (most significant byte first) or in little-endian byte order (least significant byte first).

Parameters

`out`	*z*	Pointer to the variable receiving the result half-word.
`in`	*view*	The memory view to be accessed.
`in`	*offset*	The (byte) offset of the half-word to be accessed (relative to the beginning of the view; offset zero referes to the first byte in the view.)

Returns

`true` if the read operation was successful.
`false` if the read operation failed, for examnple due to an out of bounds access or other invalid parameters. The variable pointed to by `z` must not be modified in case of an error.

A.2.2.4 size_t MemGetLength (const MemView ∗ *view*)

This function gets the length of a memory view in bytes.

Parameters

`in`	*view*	The memory view to be queried.

Returns

The length of the memory view in bytes.

A.2.2.5 bool MemGetWord (uint32_t ∗ *z*, MemView ∗ *view*, size_t *offset*)

This functions reads a single unsigned word at offset `offset` from the memory view given by `view` and stores the result in `z`. This function honors the **MEM_VIEW_BIGENDIAN** (p. 18) flag of the memory view, to determine wether the word is stored in big-endian byte order (most significant byte first) or in little-endian byte order (least significant byte first).

Parameters

`out`	*z*	Pointer to the variable receiving the result half-word.
`in`	*view*	The memory view to be accessed.
`in`	*offset*	The (byte) offset of the word to be accessed (relative to the beginning of the view; offset zero referes to the first byte in the view.)

Returns

`true` if the read operation was successful.
`false` if the read operation failed, for examnple due to an out of bounds access or other invalid parameters. The variable pointed to by `z` must not be modified in case of an error.

A.2.2.6 bool MemInit (MemView ∗ *view,* MemViewFlags *flags,* Buffer ∗ *buffer,* size_t *base,* size_t *length*)

This function initializes the memory view given by `view` for use and attaches it to the given buffer. - Additionally the function checks, that the base offset and length of the memory view denote a valid slice in the underlying buffer.

Parameters

`out`	*view*	The memory view to be initialized. On success the `view` object is initialized and ready to use. On error the state of the `view` object may be undefined.
`in`	*flags*	The flags for the new memory view. This value may be any valid combination of the flags defined in the **MemViewFlags** (p. 18) enumeration. This parameter is invalid, if any of the bits given in **MEM_VIEW_RESERVED** (p. 18) are set.
`in,out`	*buffer*	The backing store byte buffer of the memory view. (It is the API user's responsibility to ensure that the buffer remains valid, while the memory view may be in use.)
`in`	*base*	Base offset fo the first byte of the memory view within the given byte buffer.
`in`	*length*	Size of this memory view in bytes. (Number of bytes to be exposed from the underlying byte buffer).

Returns

`true` if the memory view was initialized successfully.
`false` if initialization of the memory view failed, for example due to invalid arguments. Initialization always fails if `base` and/or `length` refer to a slice outside the (current) bounds of the underlying byte buffer.

A.2.2.7 bool MemIsBigEndian (const MemView ∗ *view*)

This function tests the **MEM_VIEW_BIGENDIAN** (p. 18) flag of the given memory view to determine if the view uses big-endian byte order.

Parameters

`in`	*view*	The view to be tested.

Returns

`true` if the memory view is marked as big-endian view.
`false` if the memory view is writable (or if `view` is NULL).

A.2.2.8 bool MemIsReadOnly (const MemView ∗ *view*)

This function tests the **MEM_VIEW_READONLY** (p. 18) flag of the given memory view to determine if the view is write-protected.

Parameters

`in`	*view*	The view to be tested.

Returns

`true` if the memory view is marked as read only.
`false` if the memory view is writable (or if `view` is NULL).

A.2.2.9 bool MemSetBigEndian (MemView ∗ *view,* bool *enable*)

This view sets or clears the **MEM_VIEW_BIGENDIAN** (p. 18) flag of a given memory view. Changing the endianess mode of a memory view affects all future calls to the **MemGetWord** (p. 19), **MemSetWord** (p. 22), **MemGetHalf** (p. 19) and **MemSetHalf** (p. 21) functions.

Note

This function does NOT modify the content of a memory view. It only affects the byte order used by accessor function used later on.

Parameters

`in,out`	*view*	is the memory view to be modified.
	enable	is the new value for the **MEM_VIEW_BIGENDIAN** (p. 18) flag (`true` if the flag should be set, `false` otherwise).

Returns

`true` on success (if the mode was set).
`false` if `view` was NULL).

A.2.2.10 bool MemSetByte (MemView ∗ *view,* size_t *offset,* uint8_t *value*)

This functions stores a single byte at offset `offset` of the memory view given by `view`. The store operation fails if the storage location of the byte falls outside the bounds of the memory view or the underlying byte buffer.

This functions honors the **MEM_VIEW_READONLY** (p. 18) flag of the memory view, to determine wether the memory view is write-protected. Attempts to write to a write-protected memory view will always fail.

Parameters

`in`	*view*	The memory view to be modified.
`in`	*offset*	The offset of the byte to be stored (relative to the beginning of the view; offset zero referes to the first byte in the view.)
`in`	*value*	The value to be stored.

Returns

`true` if the read operation was successful.
`false` if the read operation failed, for examnple due to an out of bounds access or other invalid parameters.

A.2.2.11 bool MemSetHalf (MemView ∗ *view,* size_t *offset,* uint16_t *value*)

This functions stores a single unsigned half-word at offset `offset` of the memory view given by `view`. The store operation fails if the storage location of the half-word falls outside the bounds of the memory view or the underlying byte buffer.

The function honors the **MEM_VIEW_BIGENDIAN** (p. 18) flag of the memory view, to determine wether the half-word is stored in big-endian byte order (most significant byte first) or in little-endian byte order (least significant byte first).

Additionally this functions honors the **MEM_VIEW_READONLY** (p. 18) flag of the memory view, to determine wether the memory view is write-protected. Attempts to write to a write-protected memory view will always fail.

Parameters

`in`	*view*	The memory view to be modified.
`in`	*offset*	The (byte) offset of the half-word to be stored (relative to the beginning of the view; offset zero referes to the first byte in the view.)
`in`	*value*	The value to be stored.

Returns

`true` if the read operation was successful.
`false` if the read operation failed, for examnple due to an out of bounds access or other invalid parameters.

A.2.2.12 bool MemSetWord (MemView * *view,* size_t *offset,* uint32_t *value*)

This functions stores a single unsigned word at offset `offset` of the memory view given by `view`. The store operation fails if the storage location of the word falls outside the bounds of the memory view or the underlying byte buffer.

The function honors the **MEM_VIEW_BIGENDIAN** (p. 18) flag of the memory view, to determine wether the word is stored in big-endian byte order (most significant byte first) or in little-endian byte order (least significant byte first).

Additionally this functions honors the **MEM_VIEW_READONLY** (p. 18) flag of the memory view, to determine wether the memory view is write-protected. Attempts to write to a write-protected memory view will always fail.

Parameters

`in`	*view*	The memory view to be modified.
`in`	*offset*	The (byte) offset of the word to be stored (relative to the beginning of the view; offset zero referes to the first byte in the view.)
`in`	*value*	The value to be stored.

Returns

`true` if the read operation was successful.
`false` if the read operation failed, for examnple due to an out of bounds access or other invalid parameters.

A.3 Lexical token handling

Typedefs

- typedef struct **AsmToken AsmToken**

  A token produced by the lexical scanner.

Functions

- **AsmToken** ∗ **AsmTokenNew** (int type, unsigned line, unsigned column, const char ∗text, size_t text_len)

  Creates a new token.
- void **AsmTokenDelete** (**AsmToken** ∗token)

  Deletes a token object and frees any associated memory.
- unsigned **AsmTokenGetLine** (const **AsmToken** ∗token)

  Gets the source line of the given token.
- unsigned **AsmTokenGetColumn** (const **AsmToken** ∗token)

  Gets the source column of the given token.
- const char ∗ **AsmTokenGetText** (const **AsmToken** ∗token)

  Gets the textual value of the token as (read-only) C string.
- int **AsmTokenGetType** (const **AsmToken** ∗token)

  Gets the token (type) code of token.

A.3.1 Typedef Documentation

A.3.1.1 typedef struct AsmToken AsmToken

The lexical scanner (in particular the **AsmLexerScan** (p. 28)) function analyses the input stream and produces tokens for each relevant fragment. The parser then consumes the tokens to parse the assembler syntax, and ultimately to assemble the output bytecode program.

Each token contains:

- A token code (such as ASM_TOKEN_INT) to tell the parser about the nature of the input fragment.
- The textual value of the token.
- Line and column number information about the source code location of the token.

A.3.2 Function Documentation

A.3.2.1 void AsmTokenDelete (AsmToken ∗ *token*)

This function deletes a token and frees any associated memory resources. The function behaves as no-operation if a NULL token objetc is passed as `token` parameter.

Parameters

`in,out`	*token*	The token object to be disposed.

A.3.2.2 unsigned AsmTokenGetColumn (const AsmToken ∗ *token*)

This function returns the source column associated with the given token or zero, if the source column is not known.

Parameters

`in`	*token*	The token to be queried.

Returns

The source column of the token. The return value may be zero if the source line is unknown or if a NULL pointer is passed as `token` parameter.

A.3.2.3 unsigned AsmTokenGetLine (const AsmToken ∗ *token*)

This function returns the source line associated with the given token or zero, if the source line is not known.

Parameters

`in`	*token*	The token to be queried.

Returns

The source line of the token. The return value may be zero if the source line is unknown or if a NULL pointer is passed as `token` parameter.

A.3.2.4 const char∗ AsmTokenGetText (const AsmToken ∗ *token*)

This function gets the textual value of the token as (read-only) C string. The return value of this function should only be NULL if `token` is NULL. (For tokens without a textual value, the return value should be an empty string).

Parameters

`in`	*token*	The token to be queried.

Returns

A read-only NUL-terminated C string with the textual value of the token. (The returned string may be an empty string if the token does not have an associated textual value).
NULL on error (if and only if `token` is NULL).

A.3.2.5 int AsmTokenGetType (const AsmToken ∗ *token*)

This function returns the token type code of the given token. The available token type codes can be found in the `grammar.h` file, which is generated by the Lemon parser generator.

Returns

The token (type) code of the given token.
NULL on error (if and only if `token` is NULL).

A.3.2.6 AsmToken∗ AsmTokenNew (int *type*, unsigned *line*, unsigned *column*, const char ∗ *text*, size_t *text_len*)

This function creates a new token using the information provided in the parameters.

Parameters

type	is the token (type) code. Valid token type codes can be found in the `grammar.h` file, which is generated by the lemon parser generator.

This parameter is not validated by the function and will be accepted as is. (We rely on the generated parser's error handling capabilities to deal with invalid token codes, for simplicity.)

Parameters

line	is the source line number of this token (or zero if the source line is not known).
column	is the source column number of this token (should be zero if the column is not known).
text	is the textual value of the token (which may be NULL if and only if `text_len` is zero). NUL termination is optional for this parameter, since the length of this string is explicitly given by `text_len`.
text_len	is the length of the string given in `text`.

Returns

A new token object, initialized with the given parameters, on success.
NULL on error, for example in case of invalid parameters or an out of memory condition.

A.4 Lexical scanner (tokenizer)

Defines

- #define **ASM_LEXER_SUCCESS** (0)

  Lexer operation succeeded.
- #define **ASM_LEXER_EOF** (-1)

  Lexer encountered the end of file.
- #define **ASM_LEXER_ERROR** (-2)

  Lexer encountered an error.

Typedefs

- typedef struct **AsmLexer AsmLexer**

  Lexical scanner state.
- typedef int(∗ **AsmLexerReadCallback**)(void ∗app_data)

  Lexer character stream read callback.
- typedef void(∗ **AsmLexerDeleteCallback**)(void ∗app_data)

  Lexer stream dispose callback.

Functions

- **AsmLexer** ∗ **AsmLexerCreate** (**AsmLexerReadCallback** read_cb, **AsmLexerDeleteCallback** delete-_cb, void ∗app_data)

  Creates and initializes a new lexer.
- int **AsmLexerScan** (**AsmToken** ∗∗ptoken, **AsmLexer** ∗lexer)

  Scans the next token from the lexer's input stream.
- void **AsmLexerDelete** (**AsmLexer** ∗lexer)

  Deletes a lexical scanner and relases associated resources.
- unsigned **AsmLexerGetLine** (**AsmLexer** ∗lexer)

  Gets the current line number of the lexer.
- unsigned **AsmLexerGetColumn** (**AsmLexer** ∗lexer)

  Gets the current column number of the lexer.

A.4.1 Define Documentation

A.4.1.1 #define ASM_LEXER_EOF (-1)

This status code is returned by **AsmLexerScan** (p. 28) and by **AsmLexerReadCallback** (p. 27) callbacks when the end of input has been reached.

A.4.1.2 #define ASM_LEXER_ERROR (-2)

This status code is returned by **AsmLexerScan** (p. 28) and by **AsmLexerReadCallback** (p. 27) callbacks when a (potentially irrecoverable) error occurred.

A.4.1.3 #define ASM_LEXER_SUCCESS (0)

This status code is returned by **AsmLexerScan** (p. 28) when a token has been scanned successfully

A.4.2 Typedef Documentation

A.4.2.1 typedef struct AsmLexer AsmLexer

A.4.2.2 typedef void(∗ AsmLexerDeleteCallback)(void ∗app_data)

This type of callback function is passed to **AsmLexerCreate** (p. 27) as character stream dispose callback.

A.4.2.3 typedef int(∗ AsmLexerReadCallback)(void ∗app_data)

This type of callback function is passed to **AsmLexerCreate** (p. 27) as character stream read callback.

A.4.3 Function Documentation

A.4.3.1 AsmLexer∗ AsmLexerCreate (AsmLexerReadCallback *read_cb*, AsmLexerDeleteCallback *delete_cb*, void ∗ *app_data*)

This function creates and initializes a new lexer. The `read_cb` callback function is be used to read characters from input stream of the lexer. The (optional) `delete_cb` function is used to dispose the input stream when the lexer is closed (or when construction of the lexer fails.)

Parameters

read_cb	is the callback function to be used for reading characters from the input stream. This callback function is required.
delete_cb	is the optional callback function to be used for disposing the input stream when the lexer is being deleted. (This callback function will also be called, if **AsmLexerCreate** (p. 27) fails).
app_data	is an application specific callback parameter, which will be passed to all invocations of the read and delete callbacks. (**AsmLexerCreate** (p. 27) does not perform any kind of checking on this value).

Returns

The new lexer object, on success.
NULL, on error.

A.4.3.2 void AsmLexerDelete (AsmLexer ∗ *lexer*)

This function deletes a lexical scanner object and releases any assoicated resources. If present, the delete callback of the lexer will be invoked to dispose the underlying input source.

Parameters

lexer	is the lexer to be disposed.

A.4.3.3 unsigned AsmLexerGetColumn (AsmLexer ∗ *lexer*)

This function is mostly used during error reporting, to indicate the approximate location of a lexer error.

Parameters

lexer	is the lexer to be queried.

Returns

The current column number of the lexer. Column number zero should be reported if no precise line information is available.

A.4.3.4 unsigned AsmLexerGetLine (AsmLexer ∗ *lexer*)

This function is mostly used during error reporting, to indicate the approximate location of a lexer error.

Parameters

lexer	is the lexer to be queried.

Returns

The current line number of the lexer (or zero if no precise line information is available).

A.4.3.5 int AsmLexerScan (AsmToken ∗∗ *ptoken,* AsmLexer ∗ *lexer*)

Parameters

`out`	*ptoken*	is a pointer to the variable receiving the new token object. (It is the caller's responsibility to dispose the token if needed).
	lexer	is the lexcial scanner to be used.

Returns

ASM_LEXER_SUCCESS (p. 27) if a token was scanned successfully. (`*ptoken` will be a valid pointer to the scanned token).
ASM_LEXER_EOF (p. 26) if the end of file was reached without scanning a new token. (`*ptoken` will be NULL).
ASM_LEXER_ERROR (p. 26) if an error occurred while scanning for the next token. (`*ptoken` will be NULL).

A.5 TinyVM Virtual Machine Public API

B Data Structure Documentation

B.1 MemView Struct Reference

View of a region of memory.

Data Fields

- **MemViewFlags flags**

  Flags of this memory view.
- **Buffer ∗ buffer**

  Backing-store byte buffer.
- size_t **base**

  Base offset of this memory view.
- size_t **length**

  Size of this memory view.

B.1.1 Detailed Description

Memory views provide a light-weigth memory abstraction on top of byte buffers. A memory view exposes parts of the contents of a byte buffer. In addition to byte-lveel access, memory views provide functions to read and write half-word (16-bit) as well as word (32-bit) qunatities.

Memory views can either use big-endian orlittle-endian byte ordering when accessing the underlying buffers. The endianess of a memory view is defined at construction time.

The same byte buffer can be used as backing store by an arbitrary number of memory views. There NO restrictions on overlaps between views or on treating the same area of a buffer as little-endian data in one view and as big-endian data in another view.

Note

Life-time management of the memory views and their underlying byte buffers is up to the API user. It is an undetectable usage error to free a byte buffer, while it still may be used by memory views.

B.1.2 Field Documentation

B.1.2.1 size_t MemView::base

The base offset of a memory view is the offset of the first byte in the underlying byte buffer, which belongs to this memory view.

B.1.2.2 Buffer∗ MemView::buffer

This field refers to the byte buffer acting as backing store for this memory view.

Note

It is the API users responsbility to ensure that the byte buffer is not freed while the memory view may be still in use. (Such misusage can not be detected by the memory view abtsraction).

B.1.2.3 MemViewFlags MemView::flags

This field contains the bit-wise OR of one or more of the memory view flags defined in the **MemView-Flags** (p. 18) enumeration.

B.1.2.4 size_t MemView::length

The length of a memory view is the number of bytes, which are accessible via a given memory view.

The documentation for this struct was generated from the following file:

- **memview.h**

B.2 VmRegisters Struct Reference

Register of the virtual machine.

Data Fields

- uint32_t **function**

  The currently executing function.
- uint32_t **pc**

  Virtual program counter.
- uint32_t **fp**

  Virtual frame pointer.

B.2.1 Detailed Description

This data structure contains all virtual register which are required to capture the current execution state of the tinyvm virtual machine.

B.2.2 Field Documentation

B.2.2.1 uint32_t VmRegisters::fp

This field contains the stack index of the virtual frame pointer of the currently executing function. The index stored in this field is counted from the bottom of stack, and can be used in combination with **VmStackAt** (p. 65) to access elements of the (virtual) stack frame.

A virtual stack frame is laid out as follows:

fp-num_args-1	to [fp-1] contain the function arguments (the first argument is in [fp-num_args-1])
fp+0	is the function handle of the calling function
fp+1	is the (return) program counter value in the valling function.
fp+2	is the (old) frame pointer value of the calling function.
fp+3	to [fp+num_locals-1] contain the local variables.

- Slots above [fp+num_locals+0] contain the evaluation stack.

B.2.2.2 uint32_t **VmRegisters::function**

This field contains the handle of the (code) object containing the currently executing function. It may be `VM_NULL_HANDLE` if the virtual machine is currently not executing any functions.

The handle can be used with **VmGetObject** (p. 63) to fetch the **VmObject** (p. 60) corresponding to the function. This field is **VM_NULL_HANDLE** (p. 42) on startup and will be set to the entrypoint function of a bytecode program by **VmLoadByteCode** (p. 63).

B.2.2.3 uint32_t **VmRegisters::pc**

This function contains the current virtual program counter of the virtual machine. The virtual program counter is an index into the byte-code region of the currently executing function. (Program counter value zero refers to the first bytecode instruction in the function; to get the raw index of the program counter within the data-region of the function's **VmObject** (p. 60), the value **VM_FUNCTION_BYTECODE_OFFSET** (p. 41) has to be added to this value),

The **VmLoadByteCode** (p. 63) function will reset this field to zero (to start execution at the first bytecode of the entrypoint function)

The documentation for this struct was generated from the following file:

- **tinyvm.h**

C File Documentation

C.1 assembler.h File Reference

Publicly visible parts of the tinyasm assembler.

Defines

- #define **ASM_LEXER_SUCCESS** (0)

  Lexer operation succeeded.
- #define **ASM_LEXER_EOF** (-1)

  Lexer encountered the end of file.
- #define **ASM_LEXER_ERROR** (-2)

  Lexer encountered an error.

Typedefs

- typedef struct **AsmToken AsmToken**

  A token produced by the lexical scanner.
- typedef struct **AsmLexer AsmLexer**

  Lexical scanner state.
- typedef int(∗ **AsmLexerReadCallback**)(void ∗app_data)

  Lexer character stream read callback.
- typedef void(∗ **AsmLexerDeleteCallback**)(void ∗app_data)

Lexer stream dispose callback.

Functions

- **AsmToken** ∗ **AsmTokenNew** (int type, unsigned line, unsigned column, const char ∗text, size_t text_len)

  Creates a new token.
- void **AsmTokenDelete** (**AsmToken** ∗token)

  Deletes a token object and frees any associated memory.
- unsigned **AsmTokenGetLine** (const **AsmToken** ∗token)

  Gets the source line of the given token.
- unsigned **AsmTokenGetColumn** (const **AsmToken** ∗token)

  Gets the source column of the given token.
- const char ∗ **AsmTokenGetText** (const **AsmToken** ∗token)

  Gets the textual value of the token as (read-only) C string.
- int **AsmTokenGetType** (const **AsmToken** ∗token)

  Gets the token (type) code of token.
- **AsmLexer** ∗ **AsmLexerCreate** (**AsmLexerReadCallback** read_cb, **AsmLexerDeleteCallback** delete-_cb, void ∗app_data)

  Creates and initializes a new lexer.
- int **AsmLexerScan** (**AsmToken** ∗∗ptoken, **AsmLexer** ∗lexer)

  Scans the next token from the lexer's input stream.
- void **AsmLexerDelete** (**AsmLexer** ∗lexer)

  Deletes a lexical scanner and relases associated resources.
- unsigned **AsmLexerGetLine** (**AsmLexer** ∗lexer)

  Gets the current line number of the lexer.
- unsigned **AsmLexerGetColumn** (**AsmLexer** ∗lexer)

  Gets the current column number of the lexer.

C.1.1 Detailed Description

Warning

This file defines the publicly visible API of the tinyasm assembler. It as part of the assignment specification and must not be changed without explicit permission. See the `assembler_imp.h` header file for implementation details.

C.2 buffer.h File Reference

Simple byte buffer abstraction.

Typedefs

- typedef struct **Buffer Buffer**

A simple byte buffer.

Functions

- **Buffer** ∗ **BufferCreate** (void)

  Creates a new empty byte buffer.
- void **BufferFree** (**Buffer** ∗buffer)

  Frees a byte buffer and releases all associated resources.
- bool **BufferAppend** (**Buffer** ∗buffer, const void ∗data, size_t len)

  Appends data to a byte buffer.
- bool **BufferSkip** (**Buffer** ∗buffer, unsigned char filler, size_t len)

  Skips bytes at the end of a buffer.
- bool **BufferClear** (**Buffer** ∗buffer)

  Discards the content of a byte buffer.
- unsigned char ∗ **BufferGetBytes** (**Buffer** ∗buffer, size_t offset, size_t len)

  Gets a pointer into the data region of the byte buffer.
- size_t **BufferGetLength** (const **Buffer** ∗buffer)

  Gets the length of a byte buffer.

C.2.1 Detailed Description

Warning

This header file defines the public API interface of the tinyvm byte buffer API. It is part of the assignment specification and MUST NOT be changed. Details (declaration of internal functions of your implementation, defines, type definitions, ...) should go into the `buffer_imp.h` header file.

C.3 bytecodes.h File Reference

Constants and definitions of the tinyvm bytecode format.

Defines

VM constants and definitions

*The **VM_VERSION_MAJOR** (p. 53) and **VM_VERSION_MINOR** (p. 53) constants contain the latest version of the tinyvm virtual machine described in this header file. Currently the only defined version of the tinyvm architecture is version `1.0`.*

Due to the design of the bytecode instruction set, the following limits apply to bytecode programs:

- *The maximum number of local variable slots per function is restricted to **VM_MAX_LOCALS** (p. 42) (128).*
- *The maximum number of argument slots per function is restricted to **VM_MAX_ARGS** (p. 42) (128).*
- *The maximum number of result slots per function is restricted to **VM_MAX_RESULTS** (p. 42) (15).*

- *The maximum offset of a branch target relative to the branch instruction must fit into a signed (two's complement) 16-bit integer. The smallest (most negative) branch offset if* ***VM_MIN_JUMP*** *(*p. *42), the largest (most positive) branch offset if* ***VM_MAX_JUMP*** *(*p. *42).*

*The tinyvm instruction set can be encoded in big-endian or in little-endian byte order. The byte order of a bytecode blob can be detected by looking at the the first tag (****VM_TAG_UNIT*** *(*p. *53)) in the header.*

- #define **VM_BIGENDIAN** 1

  Select big-endian byte order as default for tinyvm bytecodes.
- #define **VM_VERSION_MAJOR** 0x01U

  Major version of the VM. (1)
- #define **VM_VERSION_MINOR** 0x00U

  Minor version of the VM. (0)
- #define **VM_MAX_LOCALS** 128U

  Maximum number of local variable slots of a function.
- #define **VM_MAX_ARGS** 128U

  Maximum number of argument slots of a function.
- #define **VM_MAX_RESULTS** 128U

  Maximum number of result slots of a function.
- #define **VM_MIN_JUMP** INT16_MIN

  Minimum signed jump offset for branch instructions.
- #define **VM_MAX_JUMP** INT16_MAX

  Maximum signed jump offset for branch instructions.

Object handles

The TinyVM virtual machine uses handles to refer to data and code objects within an application running on the VM. A handle is a 32-bit value which acts as unique "name" of an object in a running VM.

Assignment of handle values to data and code objects is more or less arbitrary.

- *The virtual null handle is fixed to* ***VM_NULL_HANDLE*** *(*p. *42).*
- *The tinyasm assembler by default assigns handle values in range* ***VM_DATA_HANDLE_FIRST*** *(*p. *41) to* ***VM_DATA_HANDLE_LAST*** *(*p. *41) to data objects which are defined in assembler source files via (".object data").*
- *The tinyasm assembler by default assigns handle values in range* ***VM_CODE_HANDLE_FIRST*** *(*p. *41) to* ***VM_CODE_HANDLE_LAST*** *(*p. *41) to functions (code objects) which are defined in assembler source files via (".object code" and ".function").*
- *(Almost) arbitrary handle values can be manually assigned to objects in assembler source code. It is not possible to assign the null handle to valid object; all other variants are valid.*

Note

The default ranges for code and data objects have been chosen to simplify debugging. In big-enadian bytecode programs, the automatically assigned code and data handles will start with a "0xC0" are "0xDA" byte in the hexdump, making them easy to sport in the hexdump.

- #define **VM_NULL_HANDLE** 0x00000000U

  Null handle.
- #define **VM_CODE_HANDLE_FIRST** 0xC0000000U

  First automatically assigned code object handle.
- #define **VM_CODE_HANDLE_LAST** 0xC0FFFFFFU

  Last automatically assigned code object handle.
- #define **VM_DATA_HANDLE_FIRST** 0xDA000000U

  First automatically assigned data object handle.
- #define **VM_DATA_HANDLE_LAST** 0xDAFFFFFFU

  Last automatically assigned data object handle.

Bytecode instructions

The tiny virtual machine bytecode instruction consists of seven larger functional groups:

- *Constant loading (**VM_OP_IMM32** (*p. *44), **VM_OP_HANDLE** (*p. *44))*
- *Call and branch instructions (**VM_OP_CALL** (*p. *43), **VM_OP_JMP** (*p. *45), **VM_OP_BT** (*p. *43), **VM_OP_BF** (*p. *43), **VM_OP_RET** (*p. *47))*
- *Operand stack manipulation (**VM_OP_POP** (*p. *47), **VM_OP_PEEK** (*p. *47))*
- *Compare operations (**VM_OP_CMPEQ** (*p. *44), **VM_OP_CMPNE** (*p. *44),* `VM_OP_UCMPLT`*, **VM_-OP_UCMPLE** (*p. *51), **VM_OP_UCMPGE** (*p. *51), **VM_OP_UCMPGT** (*p. *51), **VM_OP_SCMPLT** (*p. *48), **VM_OP_SCMPLE** (*p. *48), **VM_OP_SCMPGE** (*p. *48), **VM_OP_SCMPGT** (*p. *48))*
- *Load and store operations*
- #define **VM_OP_IMM32** 0x04U

  Load a (signed or unsigned) 32-bit constant (`IMM32`)
- #define **VM_OP_HANDLE** 0x05U

  Load object handle.

Calls and branches

*Function calls are performed using the standard **VM_OP_CALL** (*p. *43) instruction. There is currently no support for indirect ("function pointer") calls in the bytecode instruction set. To return from a function the **VM_OP_RET** (*p. *47) instruction is used.*

*Unconditional branches are realized with the **VM_OP_JMP** (*p. *45) instruction. The **VM_OP_BT** (*p. *43) and **VM_OP_BF** (*p. *43) instructions conditionally branch if their stack operand is non-zero (respectively zero).*

- #define **VM_OP_CALL** 0x10U

  Call function (`CALL`)
- #define **VM_OP_JMP** 0x12U

  Unconditional jump (`JMP`)
- #define **VM_OP_BT** 0x15U

  Branch if operand `a` is true (non-zero) (`BT`)

- #define **VM_OP_BF** 0x16U

  Branch if operand `a` is false (zero) (`BF`)

- #define **VM_OP_RET** 0x17U

  Return from current function. (`RET`)

Stack operations

The tinyvm virtual machine only supports two functions to directly manipulate the operand stack:

- ***VM_OP_POP*** *(p. 47) discards the top of stack operand.*
- ***VM_OP_PEEK*** *(p. 47) duplicates a specific stack element.*

- #define **VM_OP_POP** 0x20U

  Discard the `n` top-of-stack operands (`POP`)

- #define **VM_OP_PEEK** 0x21U

  Duplicates the `n-th` element of the stack (`DUPN`)

- #define **VM_OP_SWAP** 0x22U

  Swaps the two top of stack operands.

Compare instructions

*Eight compare instructions allow relational comparison of unsigned (**VM_OP_UCMPLT** (p. 51), **VM-_OP_UCMPLE** (p. 51), **VM_OP_UCMPGE** (p. 51), **VM_OP_UCMPGT** (p. 51)) and signed (**VM_OP_S-CMPLT** (p. 48), **VM_OP_SCMPLE** (p. 48), **VM_OP_SCMPGE** (p. 48), **VM_OP_SCMPGT** (p. 48)) word values.*

Each of the compare operations accepts two single word stack operands a and b and produces a single result word z which reflects the boolean outcome of the comparison. The boolean value `true` is encoded as 1, the boolean value `false` is encoded as 0.

*The result of compare operations can be used directly with conditional branches (**VM_OP_BT** (p. 43) and **VM_OP_BF** (p. 43)) and logic operations (**VM_OP_AND** (p. 42), **VM_OP_OR** (p. 47), **VM_OP_X-OR** (p. 52)) to realize complex conditional expressions.*

- #define **VM_OP_CMPEQ** 0x2EU

  Compare if operand `a` is equal to operand `b`. (`CMPEQ`)

- #define **VM_OP_CMPNE** 0x2FU

  Compare if operand `a` is not equal to operand `b`. (`CMPEQ`)

- #define **VM_OP_UCMPLT** 0x30U

  Compare if unsigned operand `a` is strictly less than unsigned operand `b`. (`UCMPLT`)

- #define **VM_OP_UCMPLE** 0x31U

  Compare if unsigned operand `a` is less or equal than unsigned operand `b`. (`UCMPLE`)

- #define **VM_OP_UCMPGE** 0x32U

  Compare if unsigned operand `a` is greater or equal than unsigned operand `b`. (`UCMPGE`)

- #define **VM_OP_UCMPGT** 0x33U

  Compare if unsigned operand `a` is strictly greater than unsigned operand `b`. (`UCMPGT`)

- #define **VM_OP_SCMPLT** 0x34U

  Compare if signed operand `a` is strictly less than signed operand `b`. (`SCMPLT`)

- #define **VM_OP_SCMPLE** 0x35U

  Compare if signed operand `a` is less or equal than signed operand `b`. (`SCMPLE`)

- #define **VM_OP_SCMPGE** 0x36U

  Compare if signed operand `a` is greater or equal than signed operand `b`. (`SCMPGE`)

- #define **VM_OP_SCMPGT** 0x37U

  Compare if signed operand `a` is strictly greater than signed operand `b`. (`SCMPGT`)

Load/store instructions

*The tiny virtual machine provides a total of six instructions for loading or storing stack values to memory locations. Local variables and call arguments can be access via the **VM_OP_LDVAR** (p. 45) and **VM_OP_STVAR** (p. 50) instructions, which statically encode the index of the variable or argument directly into the byte-code. There are no instructions to "indirectly" access local variables and arguments.*

*The content of data and code objects can be accessed via dedicated load word (**VM_OP_LDW** (p. 46)), store word (**VM_OP_STW** (p. 50)), load unsigned byte (**VM_OP_LDB** (p. 45)), store unsigned byte (**VM_OP_STB** (p. 50)) instructions.*

- #define **VM_OP_LDVAR** 0x40U

  Load from local variable or argument. (`LDVAR`)

- #define **VM_OP_STVAR** 0x41U

  Store to local variable or argument. (`STVAR`)

- #define **VM_OP_LDW** 0x42U

  Load 32-bit word from memory. (`LDW`)

- #define **VM_OP_STW** 0x43U

  Store 32-bit word to memory. (`SDW`)

- #define **VM_OP_LDB** 0x44U

  Load 8-bit byte from memory. (`LDB`)

- #define **VM_OP_STB** 0x45U

  Store 8-bit byte to memory. (`STB`)

Arithmetic and logic instructions

*The tiny virtual machine includes a standard set of arithmetic (**VM_OP_ADD** (p. 42), **VM_OP_SUB** (p. 50), **VM_OP_MUL** (p. 46), **VM_OP_SDIV** (p. 49), **VM_OP_UDIV** (p. 52), **VM_OP_NEG** (p. 46)), shift (**VM_OP_SHL** (p. 49), **VM_OP_SHR** (p. 49), **VM_OP_ASR** (p. 43), **VM_OP_ROR** (p. 48)) and logic (**VM_OP_AND** (p. 42), **VM_OP_OR** (p. 47), **VM_OP_XOR** (p. 52), **VM_OP_NOT** (p. 46)) instructions, which are encoded with a single byte.*

All arithmetic and logic instruction take two stack operands a and b on input and produce a single stack operand z on output. Stack operands are generally treated as 32-bit unsigned integers, and overflows lead to silent truncation of results.

- #define **VM_OP_ADD** 0x50U

  Add operand `a` to operand `b`. (`ADD`)

- #define **VM_OP_SUB** 0x51U

  Subtract operand `b` from operand `a`. (`SUB`)

- #define **VM_OP_MUL** 0x53U

  Multiplies operand `a` with operand `b`. (`MUL`)

- #define **VM_OP_UDIV** 0x54U

  Divide unsigned operand `a` by unsigned operand `b`. (`UDIV`)

- #define **VM_OP_SDIV** 0x55U

Divide signed operand `a` by signed operand `b`. (`SDIV`)

- #define **VM_OP_SHL** 0x56U

  Logic shift left of operand `a` by the number of bits given in operand `b`. (`SHL`)
- #define **VM_OP_SHR** 0x57U

  Logic shift right of operand `a` by the number of bits given in operand `b`. (`SHR`)
- #define **VM_OP_ASR** 0x58U

  Arithmetic shift right of operand `a` by the number of bits given in operand `b`. (`ASR`)
- #define **VM_OP_ROR** 0x59U

  Logic rotate right of operand `a` by the number of bits given in operand `b`. (`ROR`)
- #define **VM_OP_AND** 0x5AU

  Bit-wise logical AND of operand `and` operand `b`. (`AND`)
- #define **VM_OP_OR** 0x5BU

  Bit-wise inclusive OR of operand `and` operand `b`. (`OR`)
- #define **VM_OP_XOR** 0x5CU

  Bit-wise exclusive OR of operand `and` operand `b`. (`XOR`)
- #define **VM_OP_NOT** 0x5DU

  Bit-wise NOT (one's complement) of operand `a`. (`NOT`)
- #define **VM_OP_NEG** 0x5EU

  Bit-wise NEG (two's complement) of operand `a`. (`NOT`)

Virtual Machine Calls

- #define **VM_OP_VMC** 0x60U

  Virtual Machine Call (`VMC`).
- #define **VMC_CODE**(major, minor) (((major) << 4) | (minor))

  Helper macro to define virtual machine call codes.
- #define **VMC_EXIT VMC_CODE**(0, 0)

  Exit from bytecode program. `VMC(VM_EXIT)`
- #define **VMC_ASSERT VMC_CODE**(0, 1)

  Assertion in a bytecode program. `VMC(VM_ASSERT)`
- #define **VMC_READ VMC_CODE**(0, 2)

  Read a byte from the virtual input stream. `VMC(VM_READ)`
- #define **VMC_WRITE VMC_CODE**(0, 3)

  Write a byte to the virtual output stream. `VMC(VM_WRITE)`

Objects and binary format

The tiny virtual machine supports a simple binary representation for virtual machine objects, which can be used to dump the state of the virtual machine to a binary blob. The tinyasm assembler is able to produce such binary blobs.

Each element in a tinyvm virtual machine binary is prefixed with a 32-bit tag indicating the type of the element. Currently the following tags are in use:

- A single **VM_TAG_UNIT** (p. 53) indicates the start of a program unit and is accompanied by a short header giving the major and minor VM version used to encode the remaining blob.

- Zero or more **VM_TAG_OBJECT** (p. 53) tags define code and data objects which can be found in the given program unit.
- A final **VM_TAG_END** (p. 53) tag marks the end of the program unit.

An entire virtual machine binary can be described by the following pseudo-C structures:

```
struct VmBinUnit {
  uint32_t start_tag;      // Magic start tag bytes (VM_TAG_UNIT)
  uint16_t vm_major;       // Major VM version (VM_VERSION_MAJOR)
  uint16_t vm_minor;       // Minor VM version (VM_VERSION_MINOR)
  uint32_t vm_entry;       // Handle of the entry-point function
  VmBinObject[] objects;   // Objects defined in this unit
  uint8_t  end_tag[4];     // Magic end tag bytes (VM_TAG_END)
};

struct VmBinObject {
  uint32_t obj_tag;        // Magic object tag bytes (VM_TAG_OBJECT)
  uint32_t qualifiers;     // Qualifiers of this object (VmQualifiers)
  uint32_t handle;         // Handle of this object
  uint32_t length;         // Length of the object content
  uint8_t data[length];    // Payload data of the object (see remarks
   below)
};

// Valid code objects (functions) contain a small 4-byte header describing
   the
// number of arguments, local variables and results of the function. The
   payload
// data of a code object is a VmBinFunction structure:
struct VmBinFunction {
  uint8_t reserved;        // Reserved (for function flags, should be zero)
  uint8_t num_results;     // Number of result slots [0..VM_MAX_RESULTS]
  uint8_t num_args;        // Number of argument slots [0..VM_MAX_ARGS]
  uint8_t num_locals;      // Number of local slots [0..VM_MAX_LOCALS]
  uint8_t bytecode[];      // Bytecode of this object (length given by
   enclosing object)
};
```

- #define **VM_TAG_UNIT** 0x74766D00U

  Binary tag at the start of a virtual machine program unit.

- #define **VM_TAG_OBJECT** 0x6F626A00U

  Binary tag of a virtual machine object.

- #define **VM_TAG_FUNCTION** 0x66756E00U

  Binary tag of a virtual machine object.

- #define **VM_FUNCTION_NUM_RESULTS_OFFSET** 1

  Offset of the "num_results" field in a function header.

- #define **VM_FUNCTION_NUM_ARGS_OFFSET** 2

  Offset of the "num_args" field in a function header.

- #define **VM_FUNCTION_NUM_LOCALS_OFFSET** 3

  Offset of the "num_locals" field in a function header.

- #define **VM_FUNCTION_BYTECODE_OFFSET** 4

  Offset of the first bytecode instrution in a function.

- #define **VM_TAG_END** 0x656E6400U

  Binary tag at the end of a virtual machine program unit.

- enum **VmQualifiers** { **VM_QUALIFIER_NONE** = 0, **VM_QUALIFIER_DATA** = 1, **VM_QUALIFIER_CODE** = 2, **VM_QUALIFIER_INMUTABLE** = 4, **VM_QUALIFIER_PROTECTED** = 8, **VM_QUALIFIER_TYPE** = VM_QUALIFIER_DATA | VM_QUALIFIER_CODE, **VM_QUALIFIER_RESERVED** }

  Qualifiers for virtual machine objects.

C.3.1 Detailed Description

This file defines the bytecode format and instruction set of the tinyvm virtual machine. We expect your implementations to be compatible with the bytecode instruction set described hereafter.

Note

You can extend this format if you want to add your own bytecodes, as long as your implementation remains backwards compatible with the original format discussed in the assignment specification.

C.3.2 Define Documentation

C.3.2.1 #define VM_BIGENDIAN 1

This macro selects big endian byte-order for all 16-bit and 32-bit quantities. Such values are stored with their most-significant byte at the lowest memory address.

C.3.2.2 #define VM_CODE_HANDLE_FIRST 0xC0000000U

The assembler by default allocates handles for code objects (functions and objects with `code` qualifier) in the range between **VM_CODE_HANDLE_FIRST** (p. 41) and **VM_CODE_HANDLE_LAST** (p. 41).

C.3.2.3 #define VM_CODE_HANDLE_LAST 0xC0FFFFFFU

See also

VM_CODE_HANDLE_FIRST (p. 41) for details on static code handles.

C.3.2.4 #define VM_DATA_HANDLE_FIRST 0xDA000000U

The assembler by default allocates handles for code objects (functions and objects with `code` qualifier) in the range between **VM_DATA_HANDLE_FIRST** (p. 41) and **VM_DATA_HANDLE_LAST** (p. 41).

C.3.2.5 #define VM_DATA_HANDLE_LAST 0xDAFFFFFFU

See also

VM_DATA_HANDLE_FIRST (p. 41) for details on default data handle assignment.

C.3.2.6 #define VM_FUNCTION_BYTECODE_OFFSET 4

This constant gives the offset of the first bytecode instruction in the payload data of a code object. (it corresponds to virtual program counter location 0 within the function).

C.3.2.7 #define VM_FUNCTION_NUM_ARGS_OFFSET 2

This constant gives the offset of the "num_args" field in the payload data of a code object.

C.3.2.8 #define VM_FUNCTION_NUM_LOCALS_OFFSET 3

This constant gives the offset of the "num_locals" field in the payload data of a code object.

C.3.2.9 #define VM_FUNCTION_NUM_RESULTS_OFFSET 1

This constant gives the offset of the "num_results" field in the payload data of a code object.

C.3.2.10 #define VM_MAX_ARGS 128U

This constant gives the maximum number (128) of input arguments which can be used within functions implemented in byte-code. The limit is due to the use of a single byte for encoding the local index in **VM_OP_LDVAR** (p. 45) and **VM_OP_STVAR** (p. 50) instructions.

C.3.2.11 #define VM_MAX_JUMP INT16_MAX

The virtual machine jump instructions use signed (two's complement) 16-bit offsets to address the branch targets relative to the virtual program counter of the jump instruction.

This value is the largest (most positive) value which can be represented in a 16-bit two's complement integer.

C.3.2.12 #define VM_MAX_LOCALS 128U

This constant gives the maximum number (128) of local variables which can be used within function implemented in byte-code. The limit is due to the use of a single byte for encoding the local index in **VM_OP_LDVAR** (p. 45) and **VM_OP_STVAR** (p. 50) instructions.

C.3.2.13 #define VM_MAX_RESULTS 128U

This constant gives the maximum number (128) of results, which can by returned from a function implemented in byte-code.

C.3.2.14 #define VM_MIN_JUMP INT16_MIN

The virtual machine jump instructions use signed (two's complement) 16-bit offsets to address the branch targets relative to the virtual program counter of the jump instruction.

This value is the smallest (most negative) value which can be represented in a 16-bit two's complement integer.

C.3.2.15 #define VM_NULL_HANDLE 0x00000000U

This special handle value expresses the null handle. It references "no object" or equivalently the "null object". (The null handle can be is the virtual machine equivalent of a native NULL pointer)

C.3.2.16 #define VM_OP_ADD 0x50U

Bytecode

```
| 0x50 |
```

Stack

$\ldots, a, b \rightarrow \ldots z$

This instruction adds its stack operands $z = a + b$ and pushes the result (truncated to a 32-bit unsigned integer) onto the stack.

C.3.2.17 #define VM_OP_AND 0x5AU

Bytecode

```
| 0x5A |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `AND` instruction interprets both operands as unsigned 32-bit integers and performs a bit-wise AND $a\&b$ of both operands; the result z is pushed on the stack.

C.3.2.18 #define VM_OP_ASR 0x58U

Bytecode

```
| 0x58 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `ASR` instruction interprets operand a as signed 32-bit integer and operand b as as unsigned 32-bit shift quantity integers. It then performs an arithemtic shift right of operand a by b bit positions. The result z of the shift operation is pushed on the stack. Newly shifted-in bit positions are filled with the sign bit of the original a operand.

Note

Arithmetic shift right of an signed integer by n bits is equivalent to truncating division by 2^n bits. Positive values are rounded towards zero. Negative values are rounded towards $-\infty$, thus arithmetic shifting -1 by any number of bits again results in -1.

C.3.2.19 #define VM_OP_BF 0x16U

Bytecode

```
| 0x15 | off[15:8] | off[7:0] |
```

Stack

$\ldots, a \rightarrow \ldots$

The `BF` instruction unconditionally branches to another byte-code position within the current function, if the a is zero. This instruction behaves like the **VM_OP_BT** (p. 43) instruction, except that the branch condition is negated.

C.3.2.20 #define VM_OP_BT 0x15U

Bytecode

```
| 0x15 |  off[15:8] | off[7:0] |
```

Stack

$\ldots, a \rightarrow \ldots$

The `BT` instruction unconditionally branches to another byte-code position within the current function, if the a is non-zero (or if the h_a handle is different to the null handle).

The branch target is constructed by adding the signed 16-bit `off` value to the virtual program counter location of the offset field of the jump instruction.

C.3.2.21 #define VM_OP_CALL 0x10U

Bytecode

```
| 0x10 | handle[31:24] | handle[23:16] | handle[15:8] | handle[7:0] |
```

Stack

$\ldots [, arg0[, arg1[, \ldots]] \rightarrow \ldots [, ret0[, ret1, [, \ldots]]$

The `CALL` instruction calls the function identified by the function handle given in the immediate operand. The number of stack operands and return values of the call can be determined from the function header of the called function.

Call arguments are pushed onto the stack in left-to-right order, the first argument is pushed first (and thus is the "farthest" element relative to the stack pointer).

See also

VM_OP_RET (p. 47) for the return instruction.

C.3.2.22 #define VM_OP_CMPEQ 0x2EU

Bytecode

```
| 0x28 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The `CMPEQ` instruction interprets the two top of stack elements a and b as 32-bit integers and tests if a is equal to ($a = b$) b. The result of the comparison (true for false) is indicated by z.

C.3.2.23 #define VM_OP_CMPNE 0x2FU

Bytecode

```
| 0x28 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The `CMPNE` instruction interprets the two top of stack elements a and b as 32-bit integers and tests if a is not equal to ($a \neq b$) b. The result of the comparison (true for false) is indicated by z.

C.3.2.24 #define VM_OP_HANDLE 0x05U

Bytecode

```
| 0x05 | handle[31:24] | handle[23:16] | handle[15:8] | handle[7:0] |
```

Stack

$\ldots \rightarrow \ldots handle$

The `HANDLE` instruction loads an object handle onto the operand stack. This instruction behaves very similar to the **VM_OP_IMM32** (p. 44) instruction.

C.3.2.25 #define VM_OP_IMM32 0x04U

VM_OP_STB (p. 50), **VM_OP_STW** (p. 50), **VM_OP_LDVAR** (p. 45), **VM_OP_STVAR** (p. 50))

- Arithmetic and logic operations (**VM_OP_ADD** (p. 42), **VM_OP_SUB** (p. 50), **VM_OP_MUL** (p. 46), **VM_OP_UDIV** (p. 52), **VM_OP_SDIV** (p. 49), **VM_OP_SHL** (p. 49), **VM_OP_SHR** (p. 49), **VM_OP_-ASR** (p. 43), **VM_OP_ROR** (p. 48), **VM_OP_NEG** (p. 46), **VM_OP_NOT** (p. 46), **VM_OP_AND** (p. 42), **VM_OP_OR** (p. 47), **VM_OP_XOR** (p. 52))
- Virtual machine calls (**VM_OP_VMC** (p. 52))

All bytecode instructions are designed such that the type of operation and the format of ths instruction can be determined uniquely by looking at the first byte. Depending on the type and format a bytecode instruction may be 1, 2, 3 or 5 bytes long.

Bytecode

```
| 0x04 | imm[31:24] | imm[23:16] | imm[15:8] | imm[7:0] |
```

Stack

$\ldots \rightarrow \ldots, z$

The IMM32 instruction pushes an arbitrary 32-bit value z to the operand stack. Unsigned values are encoded in binary, signed values are encoded in two's complement representation. It is neither possible nor necessary to distinguish wether an IMM32 instruction encodes a signed or unsigned operand.

C.3.2.26 #define VM_OP_JMP 0x12U

Bytecode

```
| 0x12 | off[15:8] | off[7:0] |
```

Stack

$\ldots \rightarrow \ldots$

The JMP instruction unconditionally branches to another byte-code position within the current function. The branch target is constructed by adding the signed 16-bit `off` value to the virtual program counter location of the offset field of the jump instruction.

It is illegal to jump to a target outside the bounds of the current function, any attempts to branch before the beginning or past the end of a function within a byte-code program are detected and reported as errors by the VM.

C.3.2.27 #define VM_OP_LDB 0x44U

Bytecode

```
| 0x44 |
```

Stack

$\ldots, o_a, h_a \rightarrow \ldots, z$

The LDB instruction loads an unsigned 8-bit byte from the memory location denoted by `handle` and `offset`. `handle` specifies the source object for the operation, while `offset` is an offset into the object's payload data. The virtual machine must detect and report any attempts to access invalid objects or out of bound offsets. The loaded bit is zero-extended to the 32-bit unsigned result `z`.

The content of any (code or data) object can be read, unless the object has the **VM_QUALIFIER_PROTECTED** (p. 55) qualifier. Any attempts to read from a protected object should be detected and reported by the VM as error.

C.3.2.28 #define VM_OP_LDVAR 0x40U

Bytecode

```
| 0x40 | idx[7:0] |
```

Stack

$\ldots \rightarrow \ldots, z$

The LDVAR instruction loads a value z from a local variable or argument referenced by local slot index `idx`. The first local variable is at index zero, the first argument is at index **VM_MAX_LOCALS** (p. 42).

The number of local variables and arguments of the current function can be found in the function header. It is an error to access local variable or argument slot indices which are greater then the respective maximums set for the function in the function header.

C.3.2.29 #define VM_OP_LDW 0x42U

Bytecode

```
| 0x42 |
```

Stack

$\ldots, offset, handle \rightarrow \ldots, z$

The `LDW` instruction loads a 32-bit word from the memory location denoted by `handle` and `offset`. `handle` specifies the source object for the operation, while `offset` is an offset into the object's payload data. The virtual machine must detect and report any attempts to access invalid objects or out of bound offsets.

The content of any (code or data) object can be read, unless the object has the **VM_QUALIFIER_PROTECTED** (p. 55) qualifier. Any attempts to read from a protected object should be detected and reported by the VM as error.

C.3.2.30 #define VM_OP_MUL 0x53U

Bytecode

```
| 0x53 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `MUL` instruction can be used to multiply two unsigned or two signed (two's complement) operands a and b. Mixing between unsigned and two's complement numbers in one multiplication is not supported. Internally the `MUL` opcode always performs an unsigned multiplication and truncates the result to a single 32-bit word and pushes it as z result on the stack.

Note

It can be shown that the lower n-bit half of an $n \times n \rightarrow 2n$ bit multiplication can be computed in the same way for both the unsigned and the two's complement case. (Consequently there is no need to have two different mulitply instructions).

C.3.2.31 #define VM_OP_NEG 0x5EU

Bytecode

```
| 0xED |
```

Stack (value operands)

$\ldots, a \rightarrow \ldots z$

The `NEG` instruction interprets its operand as 32-bit integer and performs a unary minus (two's complement) operation; the result z is pushed on the stack.

C.3.2.32 #define VM_OP_NOT 0x5DU

Bytecode

```
| 0x5D |
```

Stack (value operands)

$\ldots, a \rightarrow \ldots z$

C.3.2.33 #define VM_OP_OR 0x5BU

Bytecode

```
| 0x5B |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `OR` instruction interprets both operands as unsigned 32-bit integers and performs a bit-wise inclusive OR $a|b$ of both operands; the result z is pushed on the stack.

C.3.2.34 #define VM_OP_PEEK 0x21U

Bytecode

```
| 0x21 | n[7:0] |
```

Stack (example for n=0)

$\ldots, a \rightarrow \ldots, a, a$

Stack (example for n=1)

$\ldots, a, b \rightarrow \ldots, a, b, a$

The `PEEK` instruction duplicates the n-th element (counted from top of stack) of the stack.

C.3.2.35 #define VM_OP_POP 0x20U

Bytecode

```
| 0x20 | n[7:0] |
```

Stack (example for n=1)

$\ldots, a \rightarrow \ldots$

Stack (example for n=2)

$\ldots, a, b \rightarrow \ldots$

The `POP` instruction discards `n` elements from at the top of stack. This operation behaves as no-operation if `n` is zero.

C.3.2.36 #define VM_OP_RET 0x17U

Bytecode

```
| 0x17 |
```

Stack

$[rets]$ (in current frame) $\rightarrow \ldots [, rets]$ (in caller's frame)

The `RET` instruction returns from the current function and passes return values to the caller. As part of the function return, the current stack frame is destroyed and execution continues in the calling function, immediately after the calling **VM_OP_CALL** (p. 43) instruction.

The number of result stack slots is encoded in the in the function header of the active function.

The local stack frame must contain **exactly** the number of result slots given in the function header, when a `RET` instruction is executed. Any attempt to return, with more or less result slots should be detected and reported as error by the VM.

C.3.2.37 #define VM_OP_ROR 0x59U

Bytecode

```
| 0x59 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `ROR` instruction interprets both operands as unsigned 32-bit integers and performs a logical rotate right of operand a by b bit positions. The result z of the shift operation is pushed on the stack. Newly shifted-in bit positions are filled with the shifted-out bit positions. The shift-amount (b) is reduced modulo 32. (Shifting operand a by any even multiple of 32 bits does not change the result value).

C.3.2.38 #define VM_OP_SCMPGE 0x36U

Bytecode

```
| 0x36 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The `SCMPGE` instruction interprets the two top of stack elements a and b as signed (two's complement) 32-bit integers and tests if a is greater or equal ($a >= b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.39 #define VM_OP_SCMPGT 0x37U

Bytecode

```
| 0x37 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The `SCMPGT` instruction interprets the two top of stack elements a and b as signed (two's complement) 32-bit integers and tests if a is strictly greater ($a > b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.40 #define VM_OP_SCMPLE 0x35U

Bytecode

```
| 0x35 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The `SCMPLE` instruction interprets the two top of stack elements a and b as signed (two's complement) 32-bit integers and tests if a is less or equal ($a <= b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.41 #define VM_OP_SCMPLT 0x34U

Bytecode

```
| 0x34 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The SCMPLT instruction interprets the two top of stack elements a and b as signed (two's complement) 32-bit integers and tests if a is strictly less ($a < b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.42 #define VM_OP_SDIV 0x55U

Bytecode

```
| 0x55 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The SDIV instruction interprets operands a and b as signed (two's complement) 32-bit integers and pushes the result z of the division a/b on the operand stack. Attempts to divide by zero, and attempts to produce a division result which overflows the 32-bit signed integer range are detected and reported as errors by the virtual machine.

C.3.2.43 #define VM_OP_SHL 0x56U

Bytecode

```
| 0x56 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The SHL instruction interprets both operands as unsigned 32-bit integers and performs a logical shift left of operand a by b bit positions. The result z of the shift operation (truncated to 32-bit) is pushed on the stack. Newly shifted-in bit positions are filled with zero bits. Shifting operand a by more than 31 bits produces a zero result.

Note

Logical shift left of an unsigned or signed integer by n bits is equivalent to multiplication by 2^n bits, assuming that no overflows occur.

C.3.2.44 #define VM_OP_SHR 0x57U

Bytecode

```
| 0x57 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The SHR instruction interprets both operands as unsigned 32-bit integers and performs a logical shift right of operand a by b bit positions. The result z of the shift operation is pushed on the stack. Newly shifted-in bit positions are filled with zero bits. Shifting operand a by more than 31 bits produces a zero result. Newly shifted-in bit positions are filled with zero bits.

Note

Logical shift right of an unsigned integer by n bits is equivalent to truncating (floor) division by 2^n bits.

C.3.2.45 #define VM_OP_STB 0x45U

Bytecode

```
| 0x45 |
```

Stack

$\ldots, a, offset, handle \rightarrow \ldots$

The `STB` instruction stores an unsigned 8-bit byte a to the memory location denoted by `handle` and `offset`. `handle` specifies the destination object for the operation, while `offset` is an offset into the object's payload data. The virtual machine must detect and report any attempts to access invalid objects or out of bound offsets.

The content of any (code or data) object can be written, unless the object has the **VM_QUALIFIER_INMUTABLE** (p. 55) qualifier. Any attempts to write to an inmutable object will be detected and reported by the VM as error.

C.3.2.46 #define VM_OP_STVAR 0x41U

Bytecode

```
| 0x41 | idx[7:0] |
```

Stack

$\ldots, a \rightarrow \ldots$

The `STVAR` instruction stores a value a found at the top of stack into the local variable or argument slot denoted by `idx`.

See also

VM_OP_LDVAR (p. 45) for more details on the indexing scheme and on error handling.

C.3.2.47 #define VM_OP_STW 0x43U

Bytecode

```
| 0x43 |
```

Stack

$\ldots, a, index, handle \rightarrow \ldots$

The `STW` instruction stores a 32-bit a to the memory location denoted by `handle` and `offset`. `handle` specifies the destination object for the operation, while `offset` is an offset into the object's payload data. The virtual machine must detect and report any attempts to access invalid objects or out of bound offsets.

The content of any (code or data) object can be written, unless the object has the **VM_QUALIFIER_INMUTABLE** (p. 55) qualifier. Any attempts to write to an inmutable object will be detected and reported by the VM as error.

C.3.2.48 #define VM_OP_SUB 0x51U

Bytecode

```
| 0x50 |
```

Stack

$$\ldots, a, b \rightarrow \ldots z$$

This instruction subtracts its stack operands $z = a - b$ and pushes the result (truncated to a 32-bit unsigned integer) onto the stack.

C.3.2.49 #define VM_OP_SWAP 0x22U

Bytecode

```
| 0x22 |
```

Stack

$$\ldots, a, b \rightarrow \ldots, b, a$$

The `SWAP` instruction swaps the two top of stack operands.

C.3.2.50 #define VM_OP_UCMPGE 0x32U

Bytecode

```
| 0x32 |
```

Stack

$$\ldots, a, b \rightarrow \ldots, z$$

The `UCMPGE` instruction interprets the two top of stack elements a and b as unsigned 32-bit integers and tests if a is greater or equal less ($a >= b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.51 #define VM_OP_UCMPGT 0x33U

Bytecode

```
| 0x33 |
```

Stack

$$\ldots, a, b \rightarrow \ldots, z$$

The `UCMPGT` instruction interprets the two top of stack elements a and b as unsigned 32-bit integers and tests if a is strictly greater ($a > b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.52 #define VM_OP_UCMPLE 0x31U

Bytecode

```
| 0x31 |
```

Stack

$$\ldots, a, b \rightarrow \ldots, z$$

The `UCMPLE` instruction interprets the two top of stack elements a and b as unsigned 32-bit integers and tests if a is less or equal less ($a <= b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.53 #define VM_OP_UCMPLT 0x30U

Bytecode

```
| 0x30 |
```

Stack

$\ldots, a, b \rightarrow \ldots, z$

The `UCMPLT` instruction interprets the two top of stack elements a and b as unsigned 32-bit integers and tests if a is strictly less ($a < b$) than b. The result of the comparison (true for false) is indicated by z.

C.3.2.54 #define VM_OP_UDIV 0x54U

Bytecode

```
| 0x54 |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `UDIV` instruction interprets operands a and b as unsigned 32-bit integers and pushes the result z of the division a/b on the operand stack. Attempts to divide by zero are detected and reported as errors by the virtual machine.

C.3.2.55 #define VM_OP_VMC 0x60U

Bytecode

```
| 0x60 | call[7:0] |
```

Stack

$\ldots [, args] \rightarrow \ldots [, rets]$ (call dependent)

Virtual machine calls provide an extension mechanism for adding up to 256 application specific instructions to the tinyvm virtual machine. Logically virtual machine calls are organized in 16 groups of up to 16 instructions per group.

The inputs and outputs of this operation depend on the actual call. The following calls are currently defined:

- **VMC_EXIT** (p. 53) (Exit from bytecode program)
- **VMC_ASSERT** (p. 53) (Assertion in a bytecode program)
- **VMC_READ** (p. 54) (Read a byte from the virtual input stream)
- **VMC_WRITE** (p. 54) (Write a byte to the virtual output stream)

Note

Virtual machine calls can be used to add additional (application specific) instructions in a simple way. The `VMC` instruction is implemented in `vm_extra.c`. See the **VMC_CODE** (p. 53) macro for a note on reserved and available call numbers.

C.3.2.56 #define VM_OP_XOR 0x5CU

Bytecode

```
| 0x5C |
```

Stack (value operands)

$\ldots, a, b \rightarrow \ldots z$

The `XOR` instruction interprets both operands as unsigned 32-bit integers and performs a bit-wise exclusive OR $a \oplus b$ of both operands; the result z is pushed on the stack.

C.3.2.57 #define VM_TAG_END 0x656E6400U

The binary representation of a program unit ends with this tag. This tag acts as terminator for the objects list contained in the program unit.

C.3.2.58 #define VM_TAG_FUNCTION 0x66756E00U

The binary representation of a virtual machine object starts with this tag.

C.3.2.59 #define VM_TAG_OBJECT 0x6F626A00U

The binary representation of a virtual machine object starts with this tag.

C.3.2.60 #define VM_TAG_UNIT 0x74766D00U

The binary representation of a program unit starts with this tag (in big-endian order). It corresponds to the byte sequence `'tvm\0'`.

C.3.2.61 #define VM_VERSION_MAJOR 0x01U

This constant defines the major version of the tiny virtual machine architecture described in this file.

C.3.2.62 #define VM_VERSION_MINOR 0x00U

This constant defines the minor version of the tiny virtual machine architecture described in this file.

C.3.2.63 #define VMC_ASSERT VMC_CODE(0, 1)

Bytecode

```
| 0x60 | 0x01 |
```

Stack

$\ldots, test \rightarrow \ldots$

The `VMC_ASSERT` virtual machine call allows bytecode programs to perfom the equivalent of a C `assert` assertions: The `test` operand is removed from the operand stack and compared to zero. If the `test` condition is non-zero, execution of the bytecode program will proceed normally (the assertion passes). If the `test` condition is zero, execution of the bytecode program stops and the **VmStep** (p. 67) function will fail with exit status -1. (the assertion failed).

C.3.2.64 #define VMC_CODE(*major*, *minor*) (((major) << 4) | (minor))

This macro constructs a call number for use as immediate operand of a **VM_OP_VMC** (p. 52) instruction. There are 16 `major` call groups each allowing for up to 16 minor calls.

The major call group 0 is reserved for the assignment specification. Major groups 1-15 can be freely used (in case you want to add your own `VMC` instructions).

Parameters

major	is the major call number.
minor	is the minor call number

C.3.2.65 #define VMC_EXIT VMC_CODE(0, 0)

Bytecode

```
| 0x60 | 0x00 |
```

Stack

$\ldots \rightarrow \ldots$

The `VMC_EXIT` virtual machine call signals the termination of the bytecode program to the tinyvm virtual machine. Execution of the bytecode program is immediately halted upon execution of this virtual machine call. (The **VmStep** (p. 67) function will return with exit status 1).

C.3.2.66 #define VMC_READ VMC_CODE(0, 2)

Bytecode

```
| 0x60 | 0x02 |
```

Stack

$\ldots \rightarrow \ldots, in$

The `VMC_READ` virtual machine call reads the next byte from the bytecode program's virtual input stream and pushes it into the operand stack. The `in` result is either the value of the read byte (0-255) or 0xFFFFFFFF if no (more) bytes are available in the virtual machine's input stream. (This virtual machine call resembles the standard C `getchar` function).

C.3.2.67 #define VMC_WRITE VMC_CODE(0, 3)

Bytecode

```
| 0x60 | 0x03 |
```

Stack

$\ldots, out \rightarrow \ldots$

The `VMC_WRITE` virtual machine call grabs the byte value to be written from the top of the operand stack and writes it to the bytecode program's virtual output stream. The `out` byte value is silently truncated to an 8-bit unsigned value. (This virtual machine call resembles the standard C `putchar` function).

C.3.3 Enumeration Type Documentation

C.3.3.1 enum VmQualifiers

Qualifiers define the type and nature of virtual machine objects. This enumeration lists the bit-masks for the individual object qualifiers.

Enumerator:

VM_QUALIFIER_NONE No qualifiers set. This value indicates that no object qualifiers are set at all. Objects with this qualifier value are invalid. Valid objects always have at least the **VM_QUALIFIER_DATA** (p. 54) or the **VM_QUALIFIER_CODE** (p. 54) qualifier set. No qualifiers

VM_QUALIFIER_DATA Object contains data. This qualifier identifies an object as data object. - Data objects can contain arbitrary bytecode program defined content. The virtual machine does not impose any structual restrictionas on the object content.

Data objects are mutually exclusive with code objects. A data object, which has the **VM_QUALIFIER_DATA** (p. 54) qualifier can not have the **VM_QUALIFIER_CODE** (p. 54) qualifier.

See also

VM_QUALIFIER_TYPE (p. 55) for a convenience bitmask to test for code and data objects.

VM_QUALIFIER_CODE Object contains code. (Function) This qualifier indicates that an object contains code for a function. The content of code objects consists of a valid virtual machine function header, followed by the bytecode of the function body.

Typically code objects also have the **VM_QUALIFIER_INMUTABLE** (p. 55) qualifier, to indicate that the function code is not modifiable.
Code objects are mutually exclusive with data objects. A code object, which has the **VM_QUALIFIER_CODE** (p. 54) qualifier can not have the **VM_QUALIFIER_DATA** (p. 54) qualifier.
See also

VM_QUALIFIER_TYPE (p. 55) for a convenience bitmask to test for code and data objects.

VM_QUALIFIER_INMUTABLE Object is inmutable (read only). The inmutable qualifier marks the content of an object as read-only. This qualifier is valid on code and data objects. Any attempts to modify the content of an inmutable object with one of the store byte-codes will by detected and reported by the virtual machine as error.

VM_QUALIFIER_PROTECTED Object is protected against readback from bytecode programs. - The unreadable qualifier marks the content of an object as not readable by bytecode instructions (**VM_OP_LDB** (p. 45), **VM_OP_LDW** (p. 46)). This qualifier is valid on code and data objects. An object qualified with this qualifier can not only by read by the VM itself using the **VmAccessObject** (p. 60) function. Any attempts to read the object with an **VM_OP_LDB** (p. 45) or **VM_OP_LDW** (p. 46) bytecode causes a virtual machine error.

VM_QUALIFIER_TYPE Mask to identify the type (code or data) of an object. This enum value defines a convenience bit-mask to simplify identification of an object as code or data object.

VM_QUALIFIER_RESERVED Mask with all reserved qualifiers. This enum value defines a bitmask with all reserved qualifiers, which can not be used in virtual machine bytecode files.

C.4 memview.h File Reference

Memory views on top of byte buffers.

Data Structures

- struct **MemView**

  View of a region of memory.

Enumerations

- enum **MemViewFlags** { **MEM_VIEW_NORMAL** = 0, **MEM_VIEW_READONLY** = 1, **MEM_VIEW_BIGENDIAN** = 2, **MEM_VIEW_RESERVED** = ~(MEM_VIEW_READONLY | MEM_VIEW_BIGENDIAN) }

  Flags for memory views.

Functions

- bool **MemInit** (**MemView** ∗view, **MemViewFlags** flags, **Buffer** ∗buffer, size_t base, size_t length)

  Initializes a memory view for use.
- size_t **MemGetLength** (const **MemView** ∗view)

  Gets the length of a memory view in bytes.
- bool **MemIsReadOnly** (const **MemView** ∗view)

  Tests if a memory view is marked as read only.
- bool **MemIsBigEndian** (const **MemView** ∗view)

Tests if a memory view uses big-endian byte-order.

- bool **MemSetBigEndian** (**MemView** ∗view, bool enable)

  Enables or disables big-endian byte-order of a memory view.
- unsigned char ∗ **MemGetBytes** (**MemView** ∗view, size_t offset, size_t length)

  Gets a raw pointer to a range of bytes in a memory view.
- bool **MemGetByte** (uint8_t ∗z, **MemView** ∗view, size_t offset)

  Reads a single byte (unsigned 8-bit integer) from a memory view.
- bool **MemSetByte** (**MemView** ∗view, size_t offset, uint8_t value)

  Writes a single byte (unsigned 8-bit integer) to a memory view.
- bool **MemGetHalf** (uint16_t ∗z, **MemView** ∗view, size_t offset)

  Reads a single unsigned half-word (16-bit integer) from a memory view.
- bool **MemSetHalf** (**MemView** ∗view, size_t offset, uint16_t value)

  Writes a single unsigned half-word (16-bit integer) to a memory view.
- bool **MemGetWord** (uint32_t ∗z, **MemView** ∗view, size_t offset)

  Reads a single unsigned word (32-bit integer) from a memory view.
- bool **MemSetWord** (**MemView** ∗view, size_t offset, uint32_t value)

  Writes a single unsigned word (32-bit integer) to a memory view.

C.4.1 Detailed Description

C.5 tinyvm.h File Reference

Public API interface of the tiny virtual machine.

Data Structures

- struct **VmRegisters**

  Register of the virtual machine.

Functions

Access to the virtual machine execution state

The complete execution state of the tinyvm virtual machine is defined by all loaded objects, the operand stack, the virtual frame pointer, the handle of the currently executing function and the current program counter.

*The **VmRegisters** (*p. *31) data structure and the **VmGetRegs** (*p. *63) and **VmSetRegs** (*p. *65) functions allow access and modification to the virtual frame pointer, the active function handle and the current program counter.*

- bool **VmGetRegs** (**VmRegisters** ∗regs, **VmContext** ∗vm)

  Gets the execution state of the virtual machine.
- bool **VmSetRegs** (**VmContext** ∗vm, const **VmRegisters** ∗regs)

  Sets the execution state of the virtual machine.

Operand stack access

- bool **VmStackPush** (**VmContext** ∗vm, const uint32_t ∗values, size_t count)

  Pushes values to the virtual machine stack.
- bool **VmStackPop** (uint32_t ∗values, **VmContext** ∗vm, size_t count)

  Pops values from the virtual machine stack.
- size_t **VmStackGetUsage** (**VmContext** ∗vm)

  Returns the number of currently used stack slots of a virtual machine.
- uint32_t ∗ **VmStackAt** (**VmContext** ∗vm, size_t index)

  Gets a pointer to a stack element, index relatively to the bottom of the stack.

Virtual machine control

- typedef struct **VmContext VmContext**

  Virtual machine context.
- typedef int(∗ **VmIoRead**)(**VmContext** ∗vm)

  Read callback of the virtual machine.
- typedef bool(∗ **VmIoWrite**)(**VmContext** ∗vm, unsigned char c)

  Creates a new tinyvm instance.
- **VmContext** ∗ **VmCreate** (size_t max_stack, void ∗appdata, **VmIoRead** ioread, **VmIoWrite** iowrite)

  Creates a new tinyvm instance.
- void **VmDelete** (**VmContext** ∗vm)

  Deletes a virtual machine instance.
- void ∗ **VmGetAppData** (**VmContext** ∗vm)

  Gets the application data pointer of a virtual machine.
- int **VmStep** (**VmContext** ∗vm)

  Executes the next bytecode instruction. (singlestep execution)
- bool **VmLoadByteCode** (**VmContext** ∗vm, **Buffer** ∗source)

  Loads bytecode from a memory buffer.
- bool **VmLoadFile** (**VmContext** ∗vm, const char ∗filename)

  Loads bytecode from a file.

Objects

From the bytecode perspective virtual machine objects can be referenced throught 32-bit integer object handles. The virtual machine APi includes functions which allow direct creation, inspection, and manipulation of objects from within native code.

- typedef struct **VmObject VmObject**

  Virtual machine objects.
- bool **VmCreateObject** (**VmContext** ∗vm, uint32_t handle, **VmQualifiers** qualifiers, const unsigned char ∗content, size_t length)

  Creates a new object in context of the given virtual machine.
- **VmObject** ∗ **VmGetObject** (**VmContext** ∗vm, uint32_t handle)

  Looks up a virtual machine object by its handle.

- uint32_t **VmGetHandle** (**VmObject** ∗obj)

  Gets the handle of a virtual machine object.
- **VmContext** ∗ **VmGetContext** (**VmObject** ∗obj)

  Gets the parent virtual machine context of an object.
- bool **VmAccessObject** (**MemView** ∗view, **VmObject** ∗obj)

  Initializes a memory view to access the content of an object.

Auxilary logging functions

- #define **VmLogError**(vm,...) **VmLog**((vm), **VM_LOG_ERROR**, __VA_ARGS__)

  Logs a VM error message.
- #define **VmLogWarn**(vm,...) **VmLog**((vm), **VM_LOG_WARNING**, __VA_ARGS__)

  Logs a VM warning message.
- #define **VmLogInfo**(vm,...) **VmLog**((vm), **VM_LOG_INFO**, __VA_ARGS__)

  Logs a VM info message.
- #define **VmLogTrace**(vm,...) **VmLog**((vm), **VM_LOG_TRACE**, __VA_ARGS__)

  Logs a VM trace message.
- #define **VmLogDebug**(vm,...) **VmLog**((vm), **VM_LOG_DEBUG**, __VA_ARGS__)

  Logs a VM debug message.
- enum **VmLogLevel** { **VM_LOG_ERROR** = 0, **VM_LOG_WARNING** = 1, **VM_LOG_INFO** = 2, **VM_L-OG_TRACE** = 3, **VM_LOG_DEBUG** = 4 }

  Message log level of the virtual machine.
- bool **VmSetLogStream** (**VmContext** ∗vm, FILE ∗stm)

  Sets the log stream of a virtual machine.
- FILE ∗ **VmGetLogStream** (**VmContext** ∗vm)

  Gets the current log stream of a virtual machine.
- bool **VmSetLogLevel** (**VmContext** ∗vm, **VmLogLevel** level)

  Sets the current log level of a virtual machine.
- **VmLogLevel** **VmGetLogLevel** (**VmContext** ∗vm)

  Gets the current log level of a virtual machine.
- void **VmLog** (**VmContext** ∗vm, **VmLogLevel** level, const char ∗message,...) __attribute__((format(printf

  Logs a diagnostic message for a virtual machine.

C.5.1 Detailed Description

C.5.2 Define Documentation

C.5.2.1 #define VmLogDebug(*vm, ...*) VmLog((vm), VM_LOG_DEBUG, __VA_ARGS__)

See also

VmLog (p. 64) for more details.

C.5.2.2 #define VmLogError(*vm*, ...) VmLog((vm), VM_LOG_ERROR, __VA_ARGS__)

See also

VmLog (p. 64) for more details.

C.5.2.3 #define VmLogInfo(*vm*, ...) VmLog((vm), VM_LOG_INFO, __VA_ARGS__)

See also

VmLog (p. 64) for more details.

C.5.2.4 #define VmLogTrace(*vm*, ...) VmLog((vm), VM_LOG_TRACE, __VA_ARGS__)

See also

VmLog (p. 64) for more details.

C.5.2.5 #define VmLogWarn(*vm*, ...) VmLog((vm), VM_LOG_WARNING, __VA_ARGS__)

See also

VmLog (p. 64) for more details.

C.5.3 Typedef Documentation

C.5.3.1 typedef struct VmContext VmContext

This data structure encapsulates an instance of the tinyvm virtual machine.

C.5.3.2 typedef int(∗ VmIoRead)(VmContext ∗vm)

This callback type is used by the the implementation of the **VMC_READ** (p. 54) virtual machine call to read the next character from the virtual machine's input stream.

Parameters

vm	is the virtual machine context doing the read.

Returns

The value of the read byte as unsigned char (0-255).
-1 if an error occurred, while reading a byte.
-2 if no more data is available to read.

C.5.3.3 typedef bool(∗ VmIoWrite)(VmContext ∗vm, unsigned char c)

Write callback of the virtual machine.

This callback type is used by the the implementation of the **VMC_WRITE** (p. 54) virtual machine call to write the next character to the virtual machine's output stream.

Parameters

vm	is the virtual machine context doing the read.
c	is the byte to be written to the output stream.

Returns

`true` on success.
`false` on error.

C.5.3.4 typedef struct VmObject VmObject

This data structure encapsulates a virtual machine object.

C.5.4 Enumeration Type Documentation

C.5.4.1 enum VmLogLevel

Enumerator:

VM_LOG_ERROR	Message is an error message.
VM_LOG_WARNING	Message is a warning message.
VM_LOG_INFO	Message is an informational message.
VM_LOG_TRACE	Message is an execution trace message.
VM_LOG_DEBUG	Messahe is a debug message.

C.5.5 Function Documentation

C.5.5.1 bool VmAccessObject (MemView ∗ *view*, VmObject ∗ *obj*)

This function initializes the memory view given in the `view` parameter, to allow access to the object's content. The memory view is initialized using **MemInit** (p. 20).

Parameters

`out`	*view*	is the memory view to be initialized by this function. The state of the memory view is undefined, if this function fails. (The caller must not attempt to call any memory view functions other than **MemInit** (p. 20) on the view, if this function fails).
	obj	is the object to be accessed.

Returns

`true` if the memory view was initialized successfully.
`false` on error.

C.5.5.2 VmContext∗ VmCreate (size_t *max_stack*, void ∗ *appdata*, VmIoRead *ioread*, VmIoWrite *iowrite*)

This functions creates and initializes a new tinyvm instance. No bytecode or data objects are loaded for the returned virtual machine object.

Parameters

max_stack	is the maximum number of stack operand stack slots the virtual machine instance will support.
appdata	is an optional application data pointer to be associated with the virtual machine. - This pointer is stored without any changes in the **VmContext** (p. 59) structure and can be retrieved using the **VmGetAppData** (p. 61) function.
ioread	is an optional callback of type **VmIoRead** (p. 59) which will be used by the **VMC_-READ** (p. 54) virtual machine call to read the virtual machine's input stream. (This parameter can be NULL, to indicate an empty input stream)

iowrite	is an optional callback of type **VmIoWrite** (p. 59) which will be used by the **VMC_-WRITE** (p. 54) virtual machine call to write to the virtual machine's output stream. (This parameter can be NULL, to discard any **VMC_WRITE** (p. 54) output)

Returns

The newly created virtual machine object on success.
NULL on error (for example if there is insufficient memory to allocate the virtual machine)

C.5.5.3 bool VmCreateObject (VmContext ∗ *vm*, uint32_t *handle*, VmQualifiers *qualifiers*, const unsigned char ∗ *content*, size_t *length*)

This function creates a new object in context of the given virtual machine. The handle for the new object is given by the `handle` parameter. The type and nature of the new object is defined by the qualifiers given in `qualifiers`. The (code or data) conrent of the new object is defined by `content` (the length is given by `length`).

Note

See the comments in `bytecodes.h (p. 34)` for more details on the contents of function and data object objects. Data objects just contain the raw data. Code objects contain a small header, followed by the bytecode of the function.

Parameters

vm	is the virtual machine for which the object will be created.
handle	is the handle of the object to be created. Object creation will fail, if the handle is either the null handle(**VM_NULL_HANDLE** (p. 42)) or if an object with the same handle already exists in this VM.
qualifiers	are the qualifiers of the new object.
content	is the initial content data of the new object. (This parameter can be NULL to create a zero-initialized object).
length	is the length of the object content.

Returns

`true` if the object was created successfully.
`false` if the object could not be created due to an error. Some possible error cases would be out of memory conditions, bad arguments, duplicate object handles, etc.

C.5.5.4 void VmDelete (VmContext ∗ *vm*)

This function deletes a virtual machine object and releases all associated resources.

Parameters

vm	the virtual machine to be deleted.

C.5.5.5 void∗ VmGetAppData (VmContext ∗ *vm*)

This functions retrieves the application specific `appdata` pointer which was passed to **VmCreate** (p. 60) when the given VM was created. The interpretation of the returned pointer is up to the application.

Parameters

vm	is the virtual machine to query.

Returns

The `appdata` pointer which was passed to **VmCreate** (p. 60) when the virtual machine was created (the returned pointer may be NULL).
NULL if `vm` is NULL.

C.5.5.6 VmContext∗ VmGetContext (VmObject ∗ *obj*)

This functions returns the parent virtual machine context of a given object. The parent context of an object is set at creation time (when the object is constructed with **VmCreateObject** (p. 61)) and stays the same over the entire lifetime of the object.

Parameters

obj	is the object to be queried.

Returns

The parent context of the object on success.
NULL if and only if `obj` is NULL.

C.5.5.7 uint32_t VmGetHandle (VmObject ∗ *obj*)

This function returns the handle of a given virtual machine object. The NULL pointer is translated to the **VM_NULL_HANDLE** (p. 42) handle.

Parameters

obj	is the object to be queried.

Returns

The handle of the given object.
NULL if and only if `NULL` was passed for `obj`.

C.5.5.8 VmLogLevel VmGetLogLevel (VmContext ∗ *vm*)

This functions gets the current log level of a virtual machine. The current log level is the numerically greatest log level, which is logged.

Parameters

vm	is the virtual machine to query, or NULL to get the default log level.

Returns

The current log level of the given virtual machine, or the default log level if NULL was given as log level.

C.5.5.9 FILE∗ VmGetLogStream (VmContext ∗ *vm*)

This function queries the current log stream of a virtual machine. Calling this function with a NULL argument returns the default log stream (stderr).

Parameters

vm	is the virtual machine to query, or NULL to get the default log stream.

Returns

The current log stream of the given virtual machine (or the default log stream if NULL was passed as argument).

C.5.5.10 VmObject∗ VmGetObject (VmContext ∗ *vm,* uint32_t *handle*)

This function searches for an object using the object's handle. On success this function returns a pointer to the virtual machine object in question. The returned **VmObject** (p. 60) pointer is guranteed to remain valid until either the virtual machine is deleted, or until the next call of **VmLoadByteCode** (p. 63) on the same virtual machine.

Parameters

vm	is the virtual machine to scan.
handle	is the handle of the virtual machine object to find.

Returns

A pointer to the **VmObject** (p. 60) structure if an object associated with the given handle was found. NULL if no object was found, if **VM_NULL_HANDLE** (p. 42) was given as `handle` parameter, or if an error occurred.

C.5.5.11 bool VmGetRegs (VmRegisters ∗ *regs,* VmContext ∗ *vm*)

This functions gets the active virtual registers of the virtual machine.

Parameters

`out`	*regs*	is the data structure reciving a copy of the current virtual register values.
`in`	*vm*	is the virtual machine to query.

Returns

`true` if the virtual register were read successfully.
`false` in case of an error.

C.5.5.12 bool VmLoadByteCode (VmContext ∗ *vm,* Buffer ∗ *source*)

This function loads a chunk of bytecode found in the `source` buffer into a virtual machine and creates appropriate function and data objects.

This operation replaces any existing bytecode, which may have been loaded earlier by the same virtual machine, clears the operand stack and sets the virtual program counter to the entrypoint of the loaded bytecode program.

Note

The data blob contained in the source buffer must be in the format produced by the tinyasm assembler. See the `vm/bytecodes.h (p. 34)` header file and `assembler/asm_output.c` for more details on the exact format.

Parameters

vm	is the destination virtual machine context.
source	is a buffer containing the chunk of bytecode to load.

Returns

`true` if the bytecode was loaded successfully.
`false` if the bytecode could not be loaded completely.

C.5.5.13 bool VmLoadFile (VmContext ∗ *vm*, const char ∗ *filename*)

This functions loads a chunk of bytecode from a normal disk file. Internally this function first fills a byte buffer with the content of the specified file and then delegates to **VmLoadByteCode** (p. 63) for the actual loading process.

Parameters

vm	is the destination virtual machine context.
filename	is the name of the input file to be loaded.

Returns

`true` if the bytecode was loaded successfully.
`false` if the bytecode could not be loaded completely.

See also

VmLoadByteCode (p. 63) for more details on this function.

C.5.5.14 void VmLog (VmContext ∗ *vm*, VmLogLevel *level*, const char ∗ *message*, *...*)

This function logs a diagnostic message for a virtual machine. It uses **VmGetLogLevel** (p. 62) and **Vm-GetLogStream** (p. 62) to determine the log level filter and log message target.

No output is produced if the log level given in `level` is more verbose than the active log level, or if the active log stream is NULL.

Parameters

vm	is the virtual machine generating the log message. This parameter can be NULL if no virtual machine object is available.

The VM parameter is used to determine the log level filter and and log stream. See **VmGetLogLevel** (p. 62) and **VmGetLogStream** (p. 62) for details on handling of NULL values for this parameter.

Parameters

level	is the log level of the message being logged. The actual log level given in `level` must be numerically less or equal than the log level report by **VmGetLogLevel** (p. 62) to produce any output.
message	is a format string for the log message, which will be directly passed to the `vfprintf` function.
...	is the format specific argument list, which will be

C.5.5.15 bool VmSetLogLevel (VmContext ∗ *vm*, VmLogLevel *level*)

This function sets a new log level stream for the given virtual machine. The log level controls the amount of log output produced by a virtual machine.

Parameters

vm	is the virtual machine to modify.
level	is the new log level for the virtual machine.

Returns

`true` if the log level was set successfully.
`false` if the log stream level couuld not be set, either because `vm` was NULL, or because `level` was not valid.

C.5.5.16 bool VmSetLogStream (VmContext ∗ *vm*, FILE ∗ *stm*)

This function sets a new log stream for the virtual machine. Ownership of the given stream remains with the caller. It is the caller's responsibility to ensure that the stream remains valid and open for the remaining lifetime of the virtual machine, or at least until a new log stream has been set successfully.

Parameters

vm	the virtual machine to modify.
stm	the new log output stream of the virtual machine. This parameter can be NULL to completely disable log output from the virtual machine.

Returns

`true` if the log stream was set successfully.
`false` if the log stream could not be set (

C.5.5.17 bool VmSetRegs (VmContext ∗ *vm*, const VmRegisters ∗ *regs*)

This functions sets the active virtual registers of the virtual machine. This function is atomic: Either all virtual register are updated or no virtual register are updated.

Parameters

`in`	*vm*	is the virtual machine to modify.
`in`	*regs*	is the data structure with the new values of the virtual registers.

Returns

`true` if the virtual register were successfully set to the values in `regs`.
`false` in case of an error.

C.5.5.18 uint32_t∗ VmStackAt (VmContext ∗ *vm*, size_t *index*)

This function returns a pointer to a stack element, indexed relative to the bottom of the operand stack. The `index` parameter gives the distance of the requested element from the bottom of the stack; index 0 refers to the bottom of the stack, the top of stack element is at index **VmStackGetUsage** (p. 65)(...)-1.

The pointer returned by this function may become invalid if the stack grows or shrinks (due to **VmStackPush** (p. 66) and **VmStackPop** (p. 66) calls).

Parameters

vm	is the virtual machine which owns the target stack.
index	is the index of the element to retrieve.

Returns

A pointer to the requested stack element. This pointer can be used to read or write the given element. It may becomes invalid if the size of the stack changes due to a push or pop operation.

C.5.5.19 size_t VmStackGetUsage (VmContext ∗ *vm*)

Parameters

`in,out`	*vm*	is the virtual machine which owns the target stack to query.

Returns

The number of stack currently used stack slots of a virtual machine.
Zero if either the stack of the given virtual machine is empty or if `NULL` is given as `vm` parameter.

C.5.5.20 bool VmStackPop (uint32_t ∗ *values,* VmContext ∗ *vm,* size_t *count*)

This function removes `count` values found at the top of virtual machine and optionally stores them in the `values` array. The order of elements corresponds to the order in which the values are removed from the stack. (The previous top element will be stored in the first position of the values array).

The `values` array can be set to NULL, if the values from the stack are not needed. A simple way to clear the operand stack (remove all elements) is:

```
...
VmStackPop(NULL, vm, VmStackGetUsage(vm));
...
```

This operation is atomic: Either all values are removed successfully, or no change is made to the stack (in case of an error).

Parameters

`out`	*values*	is an optional array receiving the values from the top of the operand stack.
`in,out`	*vm*	is the virtual machine which owns the target stack.
	count	is the number of elements to be removed from the target stack.

Returns

`true` if all `count` values in the `values` array have been removed from the operand stack (and stored in the `values` array, if a non-NULL `values` array was given).
`false` if the operation failed, for example if the stack contains less than `count` values, or if an invlaid parameter was specified. (in case of an error, no values will be removed from the stack.)

C.5.5.21 bool VmStackPush (VmContext ∗ *vm,* const uint32_t ∗ *values,* size_t *count*)

This functions pushes `count` values found in the `values` array to the operand stack of a virtual machine. Values are pushed in the order in which they appear on the array. (the first element of the array is pushed first, ...)

This operation is atomic: Either all values are pushed successfully, or no change is made to the stack (in case of an error).

Parameters

`in,out`	*vm*	is the virtual machine context which owns the target stack.
`in`	*values*	is an optional array of values to be pushed onto the operand stack. (-This parameter can be NULL to push a given number of zero slots onto the stack).
	count	is the number of values in the `values` array.

Returns

`true` if all `count` values in the `values` array have been pushed to the operand stack.
`false` if the operation failed. (in case of an error, no values will be pushed to the stack.)

C.5.5.22 int VmStep (VmContext * *vm*)

This function fetches and executes the next bytecode instruction. The return value of this function indicates, if the instruction was executed successfully, if the program completed execution, or if an error occurred.

This function is typically used in a loop, similar to the following code snippet:

```
...
VmContext *vm = ...; // Initialization omitted for brevity
...
// Execute until either the program is done or an error occurs
int status;
do {
  status = VmStep(vm);
} while (status == 0);
...
assert (status == 1 || status == -1);
...
if (status == -1) {
  ... // VM error detected
} else if (status == 1) {
  ... // Bytecode program terminated (VM(VMC_EXIT))
}
...
```

Parameters

vm	is the virtual machine executing the instruction.

Returns

+1 if the bytecode program terminated (in response to a `VM_EXIT` virtual machine call)
0 if an instruction was executed successfully, and there is more work to do.
-1 if the virtual machine encountered an error.

References

[1] *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. `http://cwe.mitre.org/top25/`.

[2] *VMWare*. `http://www.vmware.com/`.