# bare68k Documentation

***Release 0.0.0***

**Christian Vogelgsang**

**Aug 13, 2017**

# Contents:

bare68k allows you to write **m68k system emulators** in Python 2 or 3. It consists of a **CPU emulation** for 68000/68020/68EC020 provided by the Musashi engine written in native C. A **memory map** with RAM, ROM, special function is added and you can start the CPU emulation of your system. You can intercept the running code with a trap mechanism and use powerful diagnose functions,

written by Christian Vogelgsang <chris@vogelgsang.org>

under the GNU Public License V2

# CHAPTER 1

# Tutorial

This section gives you a short tutorial on how to use the *bare68k* package.

CHAPTER 2

# API

This section describes all the high level interface modules found in the *bare68k* package. The classes wrap the low-level machine interface and offer a object-oriented access with lots of convenience methods.

## The Runtime

**class** bare68k.**Runtime**(*cpu_cfg*, *mem_cfg*, *run_cfg*, *event_handler=None*, *log_channel=None*)

The runtime controls the CPU emulation run and dispatches events.

The central entry point for every system emulation done with bare68k is the Runtime. First you configure it by passing in a CPU, memory layout and runtime configuration.

Add an optional event handler to control the processing of incoming events. Configure an optional Logger to receive all incoming traces.

**Parameters**

- **cpu_cfg** (*bare68k.CPUConfig*) – the CPU configuration
- **mem_cfg** (*bare68k.MemoryConfig*) – the memory layout for the emulation
- **run_cfg** (*bare68k.RunConfig*) – runtime options
- **event_handler** (*bare68k.EventHandler*, optional) – event handler that receives all returned events from the CPU emulation. By default the *bare68k.EventHandler* is used.
- **log_channel** (logging.Logger, optional) – a logger that logs all runtime events. By default a logger with __name__ of the module is created.

**get_cpu**()

return cpu API 'object'

**get_cpu_cfg**()

access the current CPU configuration

**get_label_mgr**()
Get the label manager associated with the runtime.

If labels are enabled a real `LabelMgr` is returned. If labels are disabled then a fake `DummyLabelMgr` is available. It provides the same interface but does nothing.

**Returns** active label manager

**Return type** `LabelMgr` or `DummyLabelMgr`

**get_mem**()
return mem API 'object'

**get_mem_cfg**()
access the current memory configuration

**get_run_cfg**()
access the current run configuration

**get_with_labels**()
Check is memory labels are enabled for the runtime.

The runtime can be either configured to enable or disable memory labels via the *RunConfig*. This function returns True if labels are enabled otherwise False.

**Returns** True if labels are enabled, otherwise False

**Return type** bool

**reset**(*init_pc*, *init_sp=2048*)
reset the CPU

before you can run the CPU you have to reset it. This will write the initial SP and the initial PC to locations 0 and 4 in RAM and pulse a reset in the CPU emulation. After this operation you are free to overwrite these values again. Now proceed to call run().

**run**(*reset_end_pc=None*, *start_pc=None*, *start_sp=None*)
run the CPU until emulation ends

This is the main loop of your emulation. The CPU emulation is run and events are processed. The events are dispatched and the associated handlers are called. If a reset opcode is encountered then the execution is terminated.

Returns a RunInfo instance giving you timing information.

**set_handler**(*event_type*, *handler*)
set a custom handler for an event type and overwrite default handler

**shutdown**()
shutdown runtime

after using the runtime you have to shut it down. this frees the allocated resources. After that you can init() again for a new run

**class** bare68k.**CPUConfig**(*cpu_type=1*)
Configure the emulated CPU.

**get_addr_bus_width**()
get address bus width of selected CPU: either 24 or 32 bits

**class** bare68k.**MemoryConfig**(*auto_align=False*)
Configuration class for the memory layout of your m68k system

**check**(*ram_at_zero=True*, *max_pages=256*)
check if gurrent layout is valid

**get_num_pages**()
: return the total number of pages required to handle the given layout

**get_page_list_str**()
: return a string showing page allocation

**get_range_list**()
: return the list of memory ranges currently allocated

**class** bare68k.**RunConfig**(*catch_kb_intr=True*, *cycles_per_run=0*, *with_labels=True*, *pc_trace_size=8*, *instr_trace=False*, *cpu_mem_trace=False*, *api_mem_trace=False*)
: define the runtime configuration

**class** bare68k.**EventHandler**(*logger=None*, *snap_create=None*, *snap_formatter=None*, *instr_logger=None*, *mem_logger=None*)
: define the event handling of the runtime

**handle_aline_trap**(*event*)
: an unbound aline trap was encountered

**handle_cb_error**(*event*)
: a callback running your code raised an exception

**handle_instr_hook**(*event*)
: an instruction hook handler returned a value != None

**handle_int_ack**(*event*)
: int ack handler did return a value != None

**handle_mem_access**(*event*)
: default handler for invalid memory accesses

**handle_mem_bounds**(*event*)
: default handler for invalid memory accesses beyond max pages

**handle_mem_special**(*event*)
: a memory special handler returned a value != None

**handle_mem_trace**(*event*)
: a cpu mem trace handler returned a value != None

**handle_reset**(*event*)
: default handler for reset opcode

CHAPTER 3

# Low Level API

The low-level API of *bare68k* is found in the `bare68k.api` module. Here the direct native calls to the `machine` extension are available.

While the *bare68k.api.cpu* CPU and `bare68k.api.mem` Memory module are also used in regular code next to the high level *API*, all other low level calls are typically wrapped by the high level API.

## CPU Access

The functions allow to read and write the current CPU state. All data `d0-d7` and address registers `a0-a7` are available. Additionally, special registers like `USP` user space stack pointer and `VBR` vector base register can be read and written. The *bare68k.api.cpu* module wraps machine's CPU access functions.

`bare68k.api.cpu.`**`get_cpu_context`** **= <Mock name='mock.get_cpu_context' id='139874130674896'>**

Read the CPU context including all registers.

Read the CPU context if you want to save the entire state. You can later restore the full CPU state with *set_cpu_context()*.

**Returns** `CPUContext` native context object contains stored state

`bare68k.api.cpu.`**`set_cpu_context`** **= <Mock name='mock.set_cpu_context' id='139874130173648'>**

Set the CPU context including all registers.

After getting the CPU state with *get_cpu_context()* you can restore the state with this function.

**Parameters** **`ctx`** (`CPUContext`) – native context object with state to restore

CHAPTER 4

# Contants

Various functions in *bare68k* require a constant value. All the constants are found in the *bare68k.consts* module.

## CPU Types

bare68k.consts.**M68K_CPU_TYPE_INVALID = 0**
invalid CPU type.

bare68k.consts.**M68K_CPU_TYPE_68000 = 1**
Motorola 68000 CPU

bare68k.consts.**M68K_CPU_TYPE_68010 = 2**
Motorola 68010 CPU

bare68k.consts.**M68K_CPU_TYPE_68EC020 = 3**
Motorola 68EC20 CPU

bare68k.consts.**M68K_CPU_TYPE_68020 = 4**
Motorola 68020 CPU

bare68k.consts.**M68K_CPU_TYPE_68030 = 5**
Motorola 68030 CPU

**Note:** Supported by disassembler ONLY

bare68k.consts.**M68K_CPU_TYPE_68040 = 6**
Motorola 68040 CPU

**Note:** Supported by disassembler ONLY

## CPU Registers

### Data Registers

`bare68k.consts.`**`M68K_REG_D0`** **= 0**
Data Register D0

`bare68k.consts.`**`M68K_REG_D1`** **= 1**
Data Register D1

`bare68k.consts.`**`M68K_REG_D2`** **= 2**
Data Register D2

`bare68k.consts.`**`M68K_REG_D3`** **= 3**
Data Register D3

`bare68k.consts.`**`M68K_REG_D4`** **= 4**
Data Register D4

`bare68k.consts.`**`M68K_REG_D5`** **= 5**
Data Register D5

`bare68k.consts.`**`M68K_REG_D6`** **= 6**
Data Register D6

`bare68k.consts.`**`M68K_REG_D7`** **= 7**
Data Register D7

### Address Registers

`bare68k.consts.`**`M68K_REG_A0`** **= 8**
Address Register A0

`bare68k.consts.`**`M68K_REG_A1`** **= 9**
Address Register A1

`bare68k.consts.`**`M68K_REG_A2`** **= 10**
Address Register A2

`bare68k.consts.`**`M68K_REG_A3`** **= 11**
Address Register A3

`bare68k.consts.`**`M68K_REG_A4`** **= 12**
Address Register A4

`bare68k.consts.`**`M68K_REG_A5`** **= 13**
Address Register A5

`bare68k.consts.`**`M68K_REG_A6`** **= 14**
Address Register A6

`bare68k.consts.`**`M68K_REG_A7`** **= 15**
Address Register A7

### Special Registers

`bare68k.consts.`**`M68K_REG_PC`** **= 16**
Program Counter PC

`bare68k.consts.`**`M68K_REG_SR`** **= 17**
Status Register SR

`bare68k.consts.`**`M68K_REG_SP`** **= 18**
The current Stack Pointer (located in A7)

`bare68k.consts.`**`M68K_REG_USP`** **= 19**
User Stack Pointer USP

`bare68k.consts.`**`M68K_REG_ISP`** **= 20**
Interrupt Stack Pointer ISP

`bare68k.consts.`**`M68K_REG_MSP`** **= 21**
Master Stack Pointer MSP

`bare68k.consts.`**`M68K_REG_SFC`** **= 22**
Source Function Code SFC

`bare68k.consts.`**`M68K_REG_DFC`** **= 23**
Destination Function Code DFC

`bare68k.consts.`**`M68K_REG_VBR`** **= 24**
Vector Base Register VBR

`bare68k.consts.`**`M68K_REG_CACR`** **= 25**
Cache Control Register CACR

`bare68k.consts.`**`M68K_REG_CAAR`** **= 26**
Cache Address Register CAAR

### Virtual Registers

`bare68k.consts.`**`M68K_REG_PREF_ADDR`** **= 27**
*Virtual Reg* – Last prefetch address

`bare68k.consts.`**`M68K_REG_PREF_DATA`** **= 28**
*Virtual Reg* – Last prefetch data

`bare68k.consts.`**`M68K_REG_PPC`** **= 29**
*Virtual Reg* – Previous value in the program counter

`bare68k.consts.`**`M68K_REG_IR`** **= 30**
Instruction register IR

`bare68k.consts.`**`M68K_REG_CPU_TYPE`** **= 31**
*Virtual Reg* – Type of CPU being run

## Interrupt Ack Special Values

`bare68k.consts.`**`M68K_INT_ACK_AUTOVECTOR`** **= 4294967295**
interrupt acknowledge to perform autovectoring

`bare68k.consts.`**`M68K_INT_ACK_SPURIOUS`** **= 4294967294**
interrupt acknowledge to cause spurios irq

# Memory Flags

## Memory Range Create Flags

bare68k.consts.**MEM_FLAGS_READ = 1**
a readable region

bare68k.consts.**MEM_FLAGS_WRITE = 2**
a writeable region

bare68k.consts.**MEM_FLAGS_RW = 3**
a read/write region

bare68k.consts.**MEM_FLAGS_TRAPS = 4**
bit flag to allow a-line traps in this region

**Note:** Or this flag with the read/write flags

## Memory Access Type

bare68k.consts.**MEM_ACCESS_R8 = 17**
byte read access

bare68k.consts.**MEM_ACCESS_R16 = 18**
word read access

bare68k.consts.**MEM_ACCESS_R32 = 20**
long read access

bare68k.consts.**MEM_ACCESS_W8 = 33**
byte write access

bare68k.consts.**MEM_ACCESS_W16 = 34**
word write access

bare68k.consts.**MEM_ACCESS_W32 = 36**
long write access

bare68k.consts.**MEM_ACCESS_MASK = 255**
constant mask to filter out memory access values

## Access Function Code

bare68k.consts.**MEM_FC_MASK = 65280**
constant mask to filter out memory access function code

bare68k.consts.**MEM_FC_USER_DATA = 4352**
access of user data

bare68k.consts.**MEM_FC_USER_PROG = 4608**
access of user program

bare68k.consts.**MEM_FC_SUPER_DATA = 8448**
access of supervisor data

bare68k.consts.**MEM_FC_SUPER_PROG = 8704**
access of supervisor program

bare68k.consts.**MEM_FC_INT_ACK = 16384**
access during interrupt acknowledge

## Function Code Masks

bare68k.consts.**MEM_FC_DATA_MASK = 256**
constant mask for user or supervisor data access

bare68k.consts.**MEM_FC_PROG_MASK = 512**
constant mask for user or supervisor program access

bare68k.consts.**MEM_FC_USER_MASK = 4096**
constant mask for user data or program access

bare68k.consts.**MEM_FC_SUPER_MASK = 8192**
constant mask for supervisor data or program access

bare68k.consts.**MEM_FC_INT_MASK = 16384**
constant mask for interrupt acknowledge

## API Special Memory Operations

bare68k.consts.**MEM_ACCESS_R_BLOCK = 4352**
read memory block

bare68k.consts.**MEM_ACCESS_W_BLOCK = 4608**
write memory block

bare68k.consts.**MEM_ACCESS_R_CSTR = 8448**
read C-string

bare68k.consts.**MEM_ACCESS_W_CSTR = 8704**
write C-string

bare68k.consts.**MEM_ACCESS_R_BSTR = 12544**
read BCPL-string

bare68k.consts.**MEM_ACCESS_W_BSTR = 12800**
write BCPL-string

bare68k.consts.**MEM_ACCESS_R_B32 = 16640**
read BPCL long (shifted to left one bit)

bare68k.consts.**MEM_ACCESS_W_B32 = 16896**
write BCPL long (shifted to right one bit)

bare68k.consts.**MEM_ACCESS_BSET = 21504**
set a memory block to a value

bare68k.consts.**MEM_ACCESS_BCOPY = 25600**
copy a memory block

## Trap Create Flags

`bare68k.consts.`**TRAP_DEFAULT = 0**
a default A-Line trap, multi shot, no rts

`bare68k.consts.`**TRAP_ONE_SHOT = 1**
flag, a one shot trap, is auto-removed after invocation

`bare68k.consts.`**TRAP_AUTO_RTS = 2**
flag, automatically perform a RTS after trap processing

## CPU Events

`bare68k.consts.`**CPU_EVENT_CALLBACK_ERROR = 0**
a Python callback triggered by the CPU emulator caused an Error or Exception

`bare68k.consts.`**CPU_EVENT_RESET = 1**
a RESET opcode was encountered

`bare68k.consts.`**CPU_EVENT_ALINE_TRAP = 2**
an A-Line Trap opcode was executed

`bare68k.consts.`**CPU_EVENT_MEM_ACCESS = 3**
a memory region was accessed with invalid op.

E.g. a read-only region was written to

`bare68k.consts.`**CPU_EVENT_MEM_BOUNDS = 4**
a memory access beyond the allocated page range occurred

`bare68k.consts.`**CPU_EVENT_MEM_TRACE = 5**
a memory trace callback in Python returned some value

`bare68k.consts.`**CPU_EVENT_MEM_SPECIAL = 6**
a special range memory region was triggered and the handler returned a value

`bare68k.consts.`**CPU_EVENT_INSTR_HOOK = 7**
the instruction trace handler was triggered and returned a value

`bare68k.consts.`**CPU_EVENT_INT_ACK = 8**
interrupt acknowledge handler was triggered and returned a value

`bare68k.consts.`**CPU_EVENT_WATCHPOINT = 10**
a watchpoint was hit

`bare68k.consts.`**CPU_EVENT_TIMER = 11**
a timer fired

`bare68k.consts.`**CPU_NUM_EVENTS = 12**
total number of machine CPU events

`bare68k.consts.`**CPU_EVENT_USER_ABORT = 12**
runtime flag, user aborted run with a `KeyboardInterrupt`

`bare68k.consts.`**CPU_EVENT_DONE = 13**
runtime flag, reached end of processing.

E.g. a RESET opcode was encountered.

# CHAPTER 5

# Change Log

## 0.1.2 (2017-08-13)

- Fixed memcfg check bug
- Added names to memcfg

## 0.1.1 (2017-07-30)

- Added support for Windows build

## 0.1.0 (2017-07-26)

- First public release

# CHAPTER 6

## Features

- all emulation code written in C for fast speed
- runs on Python 2.7 and Python 3.5
- emulates CPU 68000, 68020, and 68EC020
- use a 24 or 32 bit memory map
- define memory regions for RAM and ROM with page granularity (64k)
- special memory regions that call your code for each read/write operation
- intercept m68k code by placing ALINE-opcode based traps to call your code
- event-based CPU emulation frontend does always return to Python first
- provide Python handlers for all CPU emulation events
  - RESET opcode
  - ALINE trap opcode
  - invalid memory access (e.g. write in ROM region)
  - out of memory bounds (e.g. read above memory map)
  - control interrupt acknowledgement
  - watch and break points
  - custom timers based on CPU cycles
- extensive diagnose functions
  - instruction trace
  - memory access for both CPU and Python API
  - register dump
  - memory labels to mark memory regions with arbitrary Python data
  - all bare68k components use Python logging

- rich API to configure memory and CPU state
- store/restore CPU context

# CHAPTER 7

## Installation

- use pip:

```
$ pip install bare68k
```

- use github repository:

```
$ python setup.py install
```

- use dev setup:

```
$ python setup.py develop --user
```

CHAPTER 8

# Quick Start

Here is a small code to see **bare68k** in action:

```
from bare68k import *
from bare68k.consts import *

# configure logging
runtime.log_setup()

# configure CPU: emulate a classic m68k
cpu_cfg = CPUConfig(M68K_CPU_TYPE_68000)

# now define the memory layout of the system
mem_cfg = MemoryConfig()
# let's create a RAM page (64k) starting at address 0
mem_cfg.add_ram_range(0, 1)
# let's create a ROM page (64k) starting at address 0x20000
mem_cfg.add_rom_range(2, 1)

# use a default run configuration (no debugging enabled)
run_cfg = RunConfig()

# combine everythin into a Runtime instance for your system
rt = Runtime(cpu_cfg, mem_cfg, run_cfg)

# fill in some code
PROG_BASE=0x1000
STACK=0x800
mem = rt.get_mem()
mem.w16(PROG_BASE, 0x23c0) # move.l d0,<32b_addr>
mem.w32(PROG_BASE+2, 0)
mem.w16(PROG_BASE+6, 0x4e70) # reset

# setup CPU
cpu = rt.get_cpu()
cpu.w_reg(M68K_REG_D0, 0x42)
```

```
# reset your virtual CPU to start at PROG_BASE and setup initial stack
rt.reset(PROG_BASE, STACK)

# now run the CPU emulation until an event occurrs
# here the RESET opcode is the event we are waiting for
rt.run()

# read back some memory
val = mem.r32(0)
assert val == 0x42

# finally shutdown runtime if its no longer used
# and free resources like the allocated RAM, ROM memory
rt.shutdown()
```

# CHAPTER 9

## Indices and tables

- genindex
- modindex
- search

# Python Module Index

## b

# Index

## B



## C



## E



## G



## H



## M

## R



## S



## T