# Users's Manual for CPU Simulators

Brent Seidel
Phoenix, AZ

August 28, 2025

Note that this is a draft version and not the final version for publication.

# Contents


**1 Introduction** **5**
1.1 License . . . 5

**2 How To Obtain** **6**
2.1 Dependencies . . . 6
2.1.1 Ada Libraries . . . 6
2.1.2 Other Libraries . . . 7

**3 Usage Instructions** **9**
3.1 Using Alire . . . 9
3.2 Using `gprbuild` . . . 9

**4 API Description** **10**
4.1 General . . . 10
4.1.1 Numerical Data Types . . . 10
4.1.2 Enumeration Data Types . . . 10
4.1.3 The CPU Objects . . . 11
4.1.4 The Bus Objects . . . 12
4.1.5 The I/O Objects . . . 14
4.1.6 Loading Data to Memory . . . 15
4.1.7 Running a Simulation . . . 15
4.1.8 Variants . . . 16
4.1.9 Other . . . 17
4.2 Example . . . 17
4.3 8080 Family . . . 17
4.4 68000 Family . . . 17
4.5 6502 Family . . . 17

**5 I/O Devices** **19**
5.1 Clock . . . 20
5.2 Serial Ports . . . 20
5.2.1 Basic Serial Port . . . 20
5.2.2 Single Line Telnet Port . . . 20
5.2.3 Multi-Line Telnet Port . . . 20
5.2.4 Tape . . . 20

5.3 Disk Interfaces . . . . . 20

**6 Command Line Interface** **22**
6.1 Device Names . . . . . 22
6.2 Commands . . . . . 23
6.2.1 Attach Command . . . . . 24
6.2.2 CPU Command . . . . . 24
6.2.3 Disk Commands . . . . . 25
6.2.4 Interrupt Commands . . . . . 25
6.2.5 Tape Commands . . . . . 25
6.2.6 Printer Commands . . . . . 25
6.3 Lisp Programming . . . . . 26
6.3.1 Attach . . . . . 26
6.3.2 Disk-Close . . . . . 26
6.3.3 Disk-Geom . . . . . 27
6.3.4 Disk-Open . . . . . 27
6.3.5 Go . . . . . 27
6.3.6 Halted . . . . . 28
6.3.7 Int-state . . . . . 28
6.3.8 Last-out-addr . . . . . 28
6.3.9 Last-out-data . . . . . 28
6.3.10 Memb . . . . . 29
6.3.11 Meml . . . . . 29
6.3.12 Memw . . . . . 29
6.3.13 Num-reg . . . . . 30
6.3.14 Override-in . . . . . 30
6.3.15 Print-Close . . . . . 30
6.3.16 Print-Open . . . . . 31
6.3.17 Reg-val . . . . . 31
6.3.18 Send-int . . . . . 31
6.3.19 Sim-CPU . . . . . 31
6.3.20 Sim-init . . . . . 32
6.3.21 Sim-load . . . . . 32
6.3.22 Sim-step . . . . . 32
6.3.23 Tape-Close . . . . . 33
6.3.24 Tape-Open . . . . . 33
6.3.25 Examples . . . . . 33

**Bibleography** **34**

# Chapter 1

# Introduction

This project includes simulators for some existing processors. It can be embedded into other code as a library, or used stand-alone with a command line interpreter. The intent of these simulators are to provide instruction level simulation and not hardware level. Generally, no attempt has been made to count clock cycles - instructions may not even take the same relative amount of time to execute.

Interfaces are provided to allow the simulator to be controlled by and display data on a simulated control panel (see the Pi-Mainframe (`https://github.com/BrentSeidel/Pi-Mainframe`) project. These may be stubbed out or ignored if not needed.

## 1.1 License

# Chapter 2

# How To Obtain

This package is currently available on GitHub at `https://github.com/BrentSeidel/Sim-CPU`

## 2.1 Dependencies

### 2.1.1 Ada Libraries

The following Ada packages are used:

- `Ada.Calendar`
- `Ada.Characters.Latin_1`
- `Ada.Containers.Indefinite_Ordered_Maps`
- `Ada.Direct_IO`
- `Ada.Exceptions`
- `Ada.Integer_Text_IO`
- `Ada.Streams`
- `Ada.Strings.Fixed`
- `Ada.Strings.Maps.Constants`
- `Ada.Strings.Unbounded`
- `Ada.Text_IO`
- `Ada.Text_IO.Unbounded_IO`
- `Ada.Unchecked_Conversion`

### 2.1.2 Other Libraries

This library depends on the root package `BBS` available at `https://github.com/BrentSeidel/BBS-Ada` or using alire via "`alr get bbs`" and the `BBS.Lisp` package available at `https://github.com/BrentSeidel/Ada-Lisp` or using alire via "`alr get bbs_lisp`". Packages external to this library are marked with an asterisk.

- `BBS`*
- `BBS.lisp`*
- `BBS.lisp.evaluate`*
- `BBS.lisp.strings`*
- `BBS.Sim_CPU`
- `BBS.Sim_CPU.Clock`
- `BBS.Sim_CPU.disk`
- `BBS.Sim_CPU.Example`
- `BBS.Sim_CPU.i8080`
- `BBS.Sim_CPU.Lisp`
- `BBS.Sim_CPU.m68000`
- `BBS.Sim_CPU.m68000.line_0`
- `BBS.Sim_CPU.m68000.line_1`
- `BBS.Sim_CPU.m68000.line_2`
- `BBS.Sim_CPU.m68000.line_3`
- `BBS.Sim_CPU.m68000.line_4`
- `BBS.Sim_CPU.m68000.line_5`
- `BBS.Sim_CPU.m68000.line_6`
- `BBS.Sim_CPU.m68000.line_7`
- `BBS.Sim_CPU.m68000.line_8`
- `BBS.Sim_CPU.m68000.line_9`
- `BBS.Sim_CPU.m68000.line_b`
- `BBS.Sim_CPU.m68000.line_c`
- `BBS.Sim_CPU.m68000.line_d`
- `BBS.Sim_CPU.m68000.line_e`

- `BBS.Sim_CPU.m68000.exceptions`
- `BBS.Sim_CPU.serial`
- `BBS.Sim_CPU.serial.telnet`
- `BBS.Sim_CPU.serial.mux`
- `cli`
- `cli_parse`
- `GNAT.Sockets`*

# Chapter 3

# Usage Instructions

## 3.1 Using Alire

Alire automatically handles dependencies. To use this in your project, just issue the command "`alr with bbs_simcpu`" in your project directory. To build the standalone CLI program, first obtain the cli using "`alr get simcpucli`". Change to the appropriate directory and use "`alr build`" and "`alr run`". To use the load CP/M program, use "`alr get loadcpm`". Change to the appropriate directory and use "`alr build`" and "`alr run`".

## 3.2 Using `gprbuild`

This is a library of routines intended to be used by some program. To use these in your program, edit your `*.gpr` file to include a line to `with` the path to `bbs_simcpu_noalr.gpr`. Then in your Ada code `with` in the package(s) you need and use the routines.

This is also available as a standalone program with a command line interface (CLI). This can be used for development or debugging. To build the CLI, just build the `simcli_noalr.gpr` project file, after making sure that the dependencies are in the expected places.

There is another program `loadcpm` available to create a bootable CP/M floppy disk image. It is build using the `loadcpm_noalr.gpr` project. It takes an Intel hex file image and writes it to a floppy disk image.

# Chapter 4

# API Description

## 4.1 General

Several simulators are available for use. Each simulator may also have variation. So, one simulator may provide variations for different processors in a family of processors.

Each simulator is based on an object that derives from the `simulator` object defined in the `BBS.Sim_CPU` package. A generic simulator interface is defined with some procedures or functions that must be defined by a specific simulator and some that may be defined, if needed. There are also a number of utility functions that are not expected to be overridden by a specific simulator.

The current design has memory included in the simulator instead of being an external device. Any I/O decoding is also handled inside the simulator. This means that any reading and writing of memory has to be done via routines defined by the simulator.

### 4.1.1 Numerical Data Types

Currently, processors with address and data busses up to 32 bits wide are supported. For the address bus, use the data type `addr_bus` and for the data bus, use `data_bus`. These are both defined as a 32 bit unsigned integer. Each simulator may use as many or as few of these bits as are needed. This means that if (as in most cases) a simulator doesn't define a full 4 GB of memory, the external program can try to read or write non-existent memory. This will probably cause an exception.

### 4.1.2 Enumeration Data Types

The `proc_mode` enumeration is used to indicate the processor mode. Not all modes will be used be any given processor and the meanings are processor specific. Any of these modes could be used by a processor that doesn't define multiple modes. The defined modes are:

- *PROC_NONE* - Unspecified processor mode.
- *PROC_KERN* - Kernel mode. This is usually the most privileged mode.
- *PROC_EXEC* - Executive mode. This is a little less privileged.

- *PROC_SUP* - Supervisor mode. This is a little less privileged.
- *PROC_USER* - User mode. This is usually the least privileged mode.

The `addr_type` enumeration is used to indicate the type of memory access. They may be used for controlling access to memory. The defined types are:

- *ADDR_NONE* - Unspecified address type.
- *ADDR_INTR* - Reading interrupt data, typically an interrupt vector.
- *ADDR_DATA* - Reading or writing data.
- *ADDR_INS* - Reading instructions (probably not writing).
- *ADDR_IO* - Reading or writing to I/O ports for processors that have separate I/O addresses.

The `bus_type` enumeration defines the type of bus for attaching I/O devices and bus accesses. This can either be the I/O bus (for processors with separate I/O space) or the memory bus for memory and memory mapped I/O. The defined busses are:

- *BUS_MEMORY* - Use the memory bus.
- *BUS_IO* - Use the I/O bus

The `bus_stat` enumeration is used for reporting the result of a bus transaction. The defined results are:

- *BUS_SUCC* - The transaction succeeded. This is the result that you hope you get.
- *BUS_NONE* - Nothing exists at this address. Either uninstalled memory or non-existent I/O device.
- *BUS_PROT* - A protection violation occurred. This is typically reported by a memory management unit when the `proc_mode` doesn't match the required level for the address.
- *BUS_PARITY* - A parity error occurred during the bus transaction. This would usually be used for testing error handlers.
- *BUS_ECC1* - A correctable ECC error occurred during the bus transaction. This would usually be used for testing error handlers.
- *BUS_ECC2* - An uncorrectable ECC error occurred during the bus transaction. This would usually be used for testing error handlers

### 4.1.3 The CPU Objects

The CPU object is responsible for executing instructions. Generally before using a simulator, it must be initialized, a variant selected, and a bus attached. However this is all implementation dependent.

To initialize the CPU, use:

```
procedure init(self : in out simulator) is abstract;
```

- `self` - The simulator object to initialize.

To select a CPU variant, use:

```
procedure variant(self : in out simulator; v : Natural) is abstract;
```

- `self` - The simulator object to initialize.
- `v` - The index of the variant to use. This value is defined by the specific CPU object.

Attaching a bus to a CPU also requires attaching the cpu to the bus. This gives both the CPU and the bus object a reference to each other. The routine to attach a bus to a CPU automatically calls the routine to attach the CPU the bus. To attach a bus, use (note that this is declared as null since some CPUs may not need this and thus they won't have to create a null implementation):

```
procedure attach_bus(self : in out simulator;
                     bus : BBS.Sim_CPU.bus.bus_access;
                     index : Natural) is Null;
```

- `self` - The simulator object to initialize.
- `bus` - The bus object to attach.
- `index` - The CPU index. Expected to be used for future multi-CPU simulations. Currently unused by all defined bus objects.

### 4.1.4 The Bus Objects

This bus object contains the memory for the CPU and interfaces with the I/O devices. Once the bus object is created, the main action a host program needs to do is to attach I/O devices.

```
procedure attach_io(self : in out bus;
                    io_dev : BBS.Sim_CPU.io.io_access;
                    base_addr : addr_bus;
                    which_bus : bus_type) is abstract;
```

- `self` - The bus object to initialize.
- `io_dev` - The I/O device object to attach to the bus.
- `base_addr` - The base address to use to access the device registers.
- `which_bus` - One of the defined `bus_type` values.

The bus object provides routines for accessing memory by physical or logical addresses. In some cases these will be identical. When some sort of address translation is provided, such as bank switching or a full MMU, they may be different. All memory access routines provide a `status` parameter that contains the `bus_state` the status of the operation. If `status` is not *`BUS_SUCC`*, the operation did not complete and any data returned by a read operation should be ignored.

To read from physical memory, use:

```
function readp(self : in out bus; addr : addr_bus; status : out bus_stat)
              return data_bus is abstract;
```

- `self` - The bus object to access.
- `addr` - The memory address to read.
- `status` - the `bus_status` result of the operation.
- Returns the value at the specified address.

To write to physical memory, use:

```
procedure writep(self : in out bus; addr : addr_bus; data: data_bus;
                 status : out bus_stat) is abstract;
```

- `self` - The bus object to access.
- `addr` - The memory address to read.
- `data` - The value to write to memory.
- `status` - the `bus_status` result of the operation.

To read from logical memory, use:

```
function readl(self : in out bus; addr : addr_bus; mode : proc_mode;
              addr_kind : addr_type; status : out bus_stat)
              return data_bus is abstract;
```

- `self` - The bus object to access.
- `addr` - The memory address to read.
- `mode` - The `proc_mode` for the operation.
- `addr_kind` - The `addr_type` for the operation.
- `status` - the `bus_status` result of the operation.
- Returns the value at the specified address.

To write to logical memory, use:

```
procedure writel(self : in out bus; addr : addr_bus; data: data_bus; mode : proc_mode;
                 addr_kind : addr_type; status : out bus_stat) is abstract;
```

- `self` - The bus object to access.
- `addr` - The memory address to read.
- `data` - The value to write to memory.
- `mode` - The `proc_mode` for the operation.
- `addr_kind` - The `addr_type` for the operation.
- `status` - the `bus_status` result of the operation.

### 4.1.5 The I/O Objects

I/O objects are used to simulate various I/O devices. From the viewpoint of a CPU, an I/O device consists of a contiguous range of addresses for the device registers on either the memory or I/O bus. These device registers are used to interface with and control the device. Once a device has been attached to a bus, the CPU may communicate with it via the device registers.

If a device uses interrupts, the owning CPU must be set along with an exception code. If a device uses DMA, the owning CPU must be specified (note that this may change to be the bus that the device is attached to).

To set the owning CPU, use:

```
procedure setOwner(self : in out io_device;
                   owner : BBS.Sim_CPU.CPU.sim_access);
```

- `self` - The I/O object to access.
- `owner` - The CPU object for the owning CPU.

To set the exception code, use:

```
procedure setException(self : in out io_device;
                       except : long) is abstract;
```

- `self` - The I/O object to access.
- `except` - The exception code to use. The interpretation of this code depends on the specific CPU.

To write to the device use:

```
procedure write(self : in out io_device; addr : addr_bus;
                data : data_bus) is abstract;
```

- `self` - The I/O object to access.
- `addr` - The address of the device register to access.
- `data` - The data to write to the device register.

To read from the device use:

```
function read(self : in out io_device; addr : addr_bus)
         return data_bus is abstract;
```

- `self` - The I/O object to access.
- `addr` - The address of the device register to access.
- Returns the value from the specified device register.

### 4.1.6 Loading Data to Memory

The main routines for reading and writing simulator memory are:
Called to set a memory value.

```
procedure set_mem(self : in out simulator;
                  mem_addr : addr_bus;
                  data : data_bus) is abstract;
```

- `self` - The simulator owning the memory to set.
- `mem_addr` - The memory address. This is allowed to be a 32 bit value. The simulator itself may do range limiting or wrapping (eg. only low order 16 bits are used) internally.
- `data` - The data value to write into the memory address. This is allowed to be a 32 bit value. The simulator decides how to handle this value and what value actually gets written.

Called to read a memory value.

```
function read_mem(self : in out simulator;
             mem_addr : addr_bus) return
  data_bus is abstract;
```

- `self` - The simulator owning the memory to read.
- `mem_addr` - The memory address. This is allowed to be a 32 bit value. The simulator itself may do range limiting or wrapping (eg. only low order 16 bits are used) internally.
- Returns the value of memory at the specified memory address.

The actual addressing and data bus used are defined by the specific simulator.
For loading bulk data into memory use:

```
procedure load(self : in out simulator; name : String)
   is null;
```

Its implementation is defined by the simulator. Typically it loads an Intel Hex file or Motorola S-Record file representing a memory image.

### 4.1.7 Running a Simulation

The `start` procedure is called first to specify a starting address for program execution. This is called to start simulator execution at a specific address. This is made null rather than abstract so that simulators that don't use it don't need to override it.

```
procedure start(self : in out simulator; addr : addr_bus) is null;
```

Then, each instruction is individually executed using the `run` procedure. This is called once per frame when start/stop is in the start position and run/pause is in the run position.

```
procedure run(self : in out simulator) is abstract;
```

Certain conditions or instruction can cause a simulator to halt. The `halted` and `continue_proc` routines can be used to test for this condition and clear it.

Check if simulator is halted. If a simulator doesn't define this, it will return *False*.

```
function halted(self : in out simulator) return Boolean is (False);
```

This clears the halted flag allowing processing to continue. If a simulator doesn't define this, it has no action.

```
procedure continue_proc(self : in out simulator) is null;
```

From all of this, one can write the core of a simulator as follows:

```
--
--  A bunch of other stuff including defining some
--  simulator as "sim"
--
begin
   sim.init;
   --
   --  I/O devices can be added here...
   --
   sim.load("image.hex");
   sim.start(0);  --  Start execution at address 0
   while not sim.halted loop
      sim.run;
   end loop;
end
```

### 4.1.8 Variants

Each simulator can support variants. This enables one simulator to support multiple CPUs in a family. Since the variants supported are unique to each simulator a universal data type cannot be used. Variants are identified by a `Natural` number. The following routines are used to get the number of variants supported by a simulator, the name of each variant, the currently selected variant, and to select a variant:

Called to get number of variants. This defaults to returning one if the simulator doesn't define an alternative.

```
function variants(self : in out simulator) return Natural is (1);
```

Called to get variant name. This needs to be overridden by the simulator.

```
function variant(self : in out simulator; v : Natural)
   return String is abstract;
```

Called to get current variant index. This needs to be overridden by the simulator.

```
function variant(self : in out simulator) return Natural is abstract;
```

Called to set variant. This needs to be overridden by the simulator.

```
procedure variant(self : in out simulator; v : Natural) is abstract;
```

### 4.1.9 Other

A number of other functions are defined to support I/O devices, interfaces with a front panel, and other things. The I/O support routines will be discussed more in chapter 5 on I/O devices.

## 4.2 Example

The example simulator provides an example of using the simulator object interface. Its primary purpose is to blink the lights in interesting ways in the Pi-Mainframe (`https://github.com/BrentSeidel/Pi-Mainframe`) project. There are a number of different patterns selectable. Variants are defined for "Copy Switches", "Count", "16-Bit Scan", "16-Bit Bouncer", "Fibonacci Counter", "32-Bit Scan", and "32-Bit Bouncer".

This simulator is unusual in that it has no memory defined, but instead has several registers defined that act as memory. When reading or writing memory, the address is ignored and the value returned depends on the pattern (variant) selected.

## 4.3 8080 Family

The 8080 simulator has variants defined for the 8080, 8085, and Z-80 processors. These are 8 bit processors with an 8 bit data bus and a 16 bit address bus. In addition to a memory bus, these processors also include an I/O bus with 8 bit I/O port addressing. See [1] for more information about the 8080/8085. See [6] and [5] for more information about the Z-80.

Currently, the 8080 family does not have memory mapped I/O. This may be added at some time in the future.

When loading data using the `load` routine, the specified file is assumed to be in Intel Hex format.

## 4.4 68000 Family

The 68000 simulator has variants defined for the 68000, 68008, 68010, and CPU32. Only the 68000 and 68008 are currently implemented. The 68010 and CPU32 are for future development. Internally, these are 32 bit processors with 32 bit data and 32 bit address busses. The external address and data bus sizes depend on the variant selected. See [2] and [3] for more information about the 68000 family.

The interrupt code is interpreted with the low order bits (7-0) representing the vector number and the next 8 bits (15-8) representing the priority. Interrupts with vectors that match internally defined exception vectors are ignored. Thus only interrupt vectors in the range 25-31 and 64-255 are processed.

When loading data using the `load` routine, the specified file is assumed to be in Motorola S-Record format.

## 4.5 6502 Family

The 6502 simulator currently has only a working variants defined for the 6502 processor. This is an 8 bit processors with an 8 bit data bus and a 16 bit address bus. This processor does not have

a separate I/O address space. All I/O is mapped as part of the main memory. Three interrupts are define for normal interrupts, non-maskable interrupts, and reset. See [4] for more information about the 6502.

When loading data using the `load` routine, the specified file is assumed to be in Intel Hex format.

# Chapter 5

# I/O Devices

Each I/O device is based on an object that derives from the `io_device` object defined in the `BBS.Sim_CPU` package. A generic I/O device interface is defined with some procedures or functions that must be defined by a specific I/O device and some that may be defined, if needed. There are also a number of utility functions that are not expected to be overridden by a specific I/O device.

To be used by a simulator, an I/O device must first be attached to the simulator using the `attach_io` routine. This is called to attach an I/O device to a simulator at a specific address. Bus is simulator dependent as some CPUs have separate I/O and memory space.

```
procedure attach_io(self : in out simulator;
                    io_dev : io_access;
                    base_addr : addr_bus;
                    bus : bus_type) is abstract;
```

Many of the I/O devices will have an initialization routine specific for that device. Since this may have device specific parameters, it can't be defined as part of the `io_device` object.

Along with attaching the I/O device to the simulator, the base address needs to be set in the I/O device object. The value for `base` in this routine should match the value for `base_addr` in the `attach_io` routine. Bad things may happen if they don't match.

```
procedure setBase(self : in out io_device;
                  base : addr_bus) is abstract;
```

The I/O device `read` and `write` functions are called by the simulator when an address within the devices address range is accessed. Note that the address range starts at the `base_addr` specified in the `attach_io` call and extends through the number of addresses specified in the `getSize` function. If needed, the simulator can query the I/O device for its base address using the `getBase` function. If a device needs to do DMA or interrupt a simulator, the `setOwner` procedure must be called first to give the device a reference to the simulator. For interrupts, the `setException` routine is used to define a `long` value that the device passes to the simulator on exception.

```
procedure setException(self : in out io_device;
                       except : long);
```

If a device does not use interrupts, this routine can be declared as `null`.

## 5.1 Clock

The main function of the clock device is to provide a periodic interrupt. It can be enabled or disabled and the time between interrupts can be set in $\frac{1}{10}$th of a second. The base time can be changed by editing the code. Note that the timing is only approximate and depends on many things. Also since the simulation is not running at the same speed as actual hardware the number of instructions between each interrupt will be different from actual hardware.

## 5.2 Serial Ports

Serial ports provide a way to provide a terminal interface to a simulator. Unlike real serial ports, output buffering is done by the host operating system and it is not obvious to the I/O device if the output has been completed or not. As a result, currently no status is set or interrupts generated on output. The output is assumed to complete instantly.

### 5.2.1 Basic Serial Port

The basic serial port was developed in early testing to send output to and get input from the terminal controlling the simulator. It worked well enough to test some initial concepts, but has been superseded by the single line telnet port.

### 5.2.2 Single Line Telnet Port

The single line telnet port provides a replacement for the basic serial port. This device has an `init` routine that is used to set the TCP port. Using `telnet` to connect to this port connects to the device and will allow communication with software running on the simulator. If enabled, this device can provide interrupts to the simulator when characters are received.

### 5.2.3 Multi-Line Telnet Port

The multi-line telnet port combines 8 ports into a single simulated device. The `init` routine specifies the starting TCP port. It and the next seven ports can be accessed by `telnet` to communicate with software running on the simulator. This uses a single exception and requires fewer addresses than having 8 single line telnet ports.

### 5.2.4 Tape

The "paper" tape interface is based on a serial port. From the point of view of the host program, it reads bytes and writes bytes. The read bytes come from an attached file and the written bytes are written to an attached file (note that the should be two different files). One could also imagine this as a cassette tape interface if that better fits the computer you're simulating.

## 5.3 Disk Interfaces

The initial `disk_ctrl` device was designed with 8 inch floppy disks in mind for use by CP/M. It has been extended to allow other disk geometries, but follows the model of tracks, sectors per track,

and heads for defining a disk. It currently supports 16 bit DMA, but may be extended to support 32 bit DMA. All data transfers occur between the request and the next instruction processed by the simulator.

A `hd_ctrl` device is under development that will support a simpler model. Disks are simply a linear sequence of blocks. It will support 32 bit block addressing and 32 bit DMA.

# Chapter 6

# Command Line Interface

The command line interface first requests which simulator to use. The choices are currently 8080 and 68000. This will change as more simulators are added. Depending on which simulator is selected, a number of I/O devices are attached to the simulator. Currently, changing this will require editing and rebuilding the program.

## 6.1 Device Names

Each device has a name that can be used to refer to it. The name consists of an alphabetic device code followed by a decimal unit number. If no unit number is specified, 0 is assumed. A unit number of zero is special. It is used to refer to a generic device, such as when attaching a device - the unit number is assigned during the attachment. The currently defined device codes are (note that not all are implemented in the CLI, but will eventually be):

**CLK** – A clock device to generate a periodic interrupt.

**CON** – A console serial device. In most cases, this should not be used.

**FD** – Main disk controller with a track and sector interface. It started out as a floppy disk controller, but geometries can be defined beyond that.

**HD** – An experimental disk controller using just block numbers. It is not yet implemented and tested.

**MUX** – An 8 channel terminal multiplexer using a telnet interface.

**PRN** – A simple printer output device.

**PTP** – A paper tape interface. It can read and write simulated tapes.

**TEL** – A single channel terminal device using a telnet interface.

## 6.2 Commands

A number of commands are provided to allow the user to run and interact with software running on the simulator. The commands and their exact operation may change depending on feedback from usage.

**;** – A comment. Everything following on the line is ignored, except if the rest of the line is being used as a file name.

**ATTACH** <**device**> <**addr**> <**bus**> [<**dev-specific**> ] – Attaches a specific I/O device at the bus address. Depending on the device, device specific parameters may be required.

**BREAK** <**addr**> – Sets a breakpoint at the specified address.

**CONTINUE** – Clear a simulation halted condition. This can be abbreviated to `C`.

**CPU** <**cpu-name**> – Set the CPU to be simulated. This can only be done once per session and must be done before any command that requires a CPU.

**DEP** <**addr**> <**byte**> – Deposit a byte into memory.

**DISK** – Commands related to floppy disks and image files (see below for more details).

**DUMP** <**addr**> – Dump a block of memory starting at the specified address. This can be abbreviated to `D`.

**EXIT** – Exit the simulation (synonym for `QUIT`).

**GO** <**addr**> – Sets the execution address.

**INTERRUPT** – Can be abbreviated to `INT`. Used to enable or disable interrupt processing by the simulated CPU.

**LISP** [<**filename**> ] – Starts the Tiny-Lisp environment. If a filename is specified, Lisp commands come from that file.

**LIST** [<**devname**> ] – Lists all the I/O devices attached to the simulator. If <devname> is supplied, it provides information for that specific device.

**LOAD** <**filename**> – Loads memory with the specified file (typically Intel Hex or S-Record format).

**TAPE** – Commands relating to printer and attached file (see below for more details).

**QUIT** – Exit the simulation (synonym for `EXIT`).

**REG** – Display the registers and their contents.

**RESET** – Calls the simulator's `init` routine.

**RUN** – Runs the simulator. While running, it can be interrupted by a control-E character. This can be abbreviated to `R`.

**STEP** – Execute a single instruction, if the simulation is not halted.

**TAPE** – Commands relating to paper (or cassette) tape and attached files (see below for more details).

**TRACE <value>** – Sets the trace value. The value is interpreted by the simulator and typically causes certain information to be printed while processing.

**UNBREAK <addr>** – Clears a breakpoint at the specified address. For some simulators, the address is ignored and may not need to be specified. In this case, all or the only breakpoint is cleared.

### 6.2.1 Attach Command

The attach command is used to attach a device to the current CPU simulator. The device is specified as a generic device name. The address is given in decimal. The bus is:

**IO** – Use the I/O bus (only available on some CPUs).

**MEM** – Use memory mapped I/O (only available on some CPUs).

Some of the devices take additional parameters.

**CLK** – Takes one additional decimal value for the exception code to be presented to the CPU when an interrupt occurs.

**FD** – Takes one additional decimal parameter for the number of attached drives in the range 0-15, though 0 drives is rather pointless.

**HD** – Not yet implemented for ATTACH (experimental - under development).

**MUX** – Takes two additional decimal parameters. The first is the TCP/IP port for the first terminal (the additional terminals are assigned sequential port numbers). The second is the exception code to be presented to the CPU when an interrupt occurs.

**PRN** – Takes no additional parameters..

**PTP** – Takes no additional parameters.

**TEL** – Takes two additional decimal parameters. The first is the TCP/IP port for the first terminal (the additional terminals are assigned sequential port numbers). The second is the exception code to be presented to the CPU when an interrupt occurs.

### 6.2.2 CPU Command

The CPU command takes the name of a CPU to simulate. The following are valid CPU names: “8080”. “8085”, “Z80”, “68000”, “68008”, and “6502”.

### 6.2.3 Disk Commands

A number of subcommands are available for interfacing with simulated floppy disks and image files. The DISK command takes the form:

- DISK <controller> <subcommand> <parameters for subcommand>

The controller is the device name of an attached disk controller. The subcommands are:

**CLOSE <drive number>** – Closes the image file associated with the specified drive number.

**OPEN <drive number> <file name>** – Attaches the specified file to the specified drive.

**READONLY <drive number>** – Sets the specified drive to read-only.

**READWRITE <drive number>** – Sets the specified drive to read-write.

### 6.2.4 Interrupt Commands

A number of subcommands are available for interfacing with interrupts.

**ON** – Enables interrupt processing by the simulator (if supported).

**OFF** – Disables interrupt processing by the simulator. This may be useful for single-stepping through some code without getting interrupted by the clock or other devices.

**SEND <interrupt number>** – Sends the specified interrupt number to the simulator (if supported).

### 6.2.5 Tape Commands

The tape interface consists of a reader called `RDR` and a writer called `PUN` (for punch). Files can be opened or closed for either of these devices. The TAPE command takes the form:

- TAPE <controller> <subcommand> <parameters for subcommand>

The controller is the device name of an attached tape controller. The subcommands are:

**CLOSE <PUN or RDR>** – Closes the file attached to the reader or punch.

**OPEN <PUN or RDR> <file name>** – Attaches the specified file to the reader or punch.

### 6.2.6 Printer Commands

The printer device simply writes data sent to it to an output file. If no file is open, the data is discarded. The PRINT command takes the form:

- PRINT <controller> <subcommand> <parameters for subcommand>

The controller is the device name of an attached printer controller. The subcommands are:

**CLOSE** – Closes the file attached to the printer.

**OPEN <file name>** – Attaches the specified file to the printer.

## 6.3 Lisp Programming

The Lisp environment is based on Ada-Lisp, available at `https://github.com/BrentSeidel/Ada-Lisp`, with a few commands added to interface with a simulator. Refer to the Ada-Lisp documentation for details about the core language. This section just describes the additional commands.

### 6.3.1 Attach

#### Inputs

The following inputs are expected:

- A string representing the device name.
- An integer for the device address.
- A string representing the address bus used ("IO" or "MEM").
- Additional items specific to the device may be required.

#### Outputs

None

#### Description

Attaches an I/O device at the specified address and address bus. Refer to the Attach command (section 6.2.1) for more information.

### 6.3.2 Disk-Close

The following inputs are expected:

- A string representing the disk controller name.
- An integer for the drive number.

#### Outputs

None

#### Description

Closes the file attached to the specified disk drive.

### 6.3.3 Disk-Geom

The following inputs are expected:

- A string representing the disk controller name.
- An integer for the drive number.
- A string representing the geometry (currently "IBM" for 8 inch diskettes or "HD" for a 5MB hard disk). It is planned to extend this to allow a list specifying arbitrary geometry as well.

#### Outputs

None

#### Description

Sets the disk geometry (tracks, sectors, heads) of the specified disk drive.

### 6.3.4 Disk-Open

The following inputs are expected:

- A string representing the disk controller name.
- An integer for the drive number.
- A string representing the file name to attach to the disk drive.

#### Outputs

None

#### Description

Opens a file and attaches it to the specified disk drive.

### 6.3.5 Go

#### Inputs

A number representing the starting address for program execution.

#### Outputs

None

#### Description

Sets the starting address for program execution. The next `(sim-step)` command executes the instruction at this location.

### 6.3.6 Halted

**Inputs**

An optional boolean that if set true, clears the halted state of the simulator. If set false, does nothing.

**Outputs**

If no input is provided, it returns a boolean representing the halted state,

**Description**

This is used to clear to test the halted state of a simulator.

### 6.3.7 Int-state

**Inputs**

None

**Outputs**

A simulator dependent integer representing the state of interrupts.

**Description**

Returns a simulator dependent integer representing the state of the interrupts. Typically it is something like 0 for interrupts disabled and 1 for interrupts enabled. For systems with priorities, it may be the processor priority.

### 6.3.8 Last-out-addr

**Inputs**

None

**Outputs**

The address used by the last output instruction.

**Description**

Returns the address used by the last output instruction. This is only meaningful if the processor being simulated has a separate output bus rather than memory mapped I/O.

### 6.3.9 Last-out-data

**Inputs**

None

**Outputs**

The data used by the last output instruction.

**Description**

The data used by the last output instruction. This is only meaningful if the processor being simulated has a separate output bus rather than memory mapped I/O.

### 6.3.10 Memb

**Inputs**

A memory address and an optional byte value.

**Outputs**

If no byte value is provided, it returns the byte at the memory address.

**Description**

Reads or writes a byte value in memory.

### 6.3.11 Meml

**Inputs**

A memory address and an optional long value.

**Outputs**

If no long value is provided, it returns the long at the memory address.

**Description**

Reads or writes a long value in memory.

### 6.3.12 Memw

**Inputs**

A memory address and an optional word value.

**Outputs**

If no word value is provided, it returns the word at the memory address.

**Description**

Reads or writes a word value in memory.

### 6.3.13 Num-reg

#### Inputs

None

#### Outputs

The number of registers defined by the simulator.

#### Description

Returns the number of registers defined by the simulator. This could be used for iterating through the registers and displaying their values.

### 6.3.14 Override-in

#### Inputs

Two integers representing the address of an I/O port and the data to provide.

#### Outputs

None

#### Description

Provides an address and data to override the next input instruction. If the instruction reads from the address, the provided data is read by the simulator instead of reading from the actual I/O device. Inputs from a different address work normally and any input instruction clears the override. This is intended for testing purposes.

### 6.3.15 Print-Close

The following inputs are expected:

- A string representing the printer controller name.

#### Outputs

None

#### Description

Closes the file attached to the specified printer.

### 6.3.16 Print-Open

The following inputs are expected:

- A string representing the printer controller name.
- A string representing the file name to attach to the printer.

#### Outputs

None

#### Description

Opens a file and attaches it to the specified printer.

### 6.3.17 Reg-val

#### Inputs

A register number

#### Outputs

The value of the specified register, or an error if the register number is not in range.

#### Description

Returns the value of the specified register.

### 6.3.18 Send-int

#### Inputs

An integer interrupt code.

#### Outputs

None.

#### Description

Sends the specified interrupt code to the simulator for processing. The processing done depends on the specific simulator. It may be ignored, be a vector number, be an instruction, or something else.

### 6.3.19 Sim-CPU

#### Inputs

A string containing the CPU name. The following are valid CPU names: "8080". "8085", "Z80", "68000", "68008", and "6502".

**Outputs**

None

**Description**

Selects the CPU to simulate. This can only be done once and must be done before any command that attempts to access the CPU.

### 6.3.20 Sim-init

**Inputs**

None.

**Outputs**

None.

**Description**

Calls the initialization routine to the current simulator. The action is simulator dependent, but typically clears the registers. The memory may, or may not be cleared.

### 6.3.21 Sim-load

**Inputs**

A string containing a file name.

**Outputs**

None.

**Description**

Calls the load routine for the current simulator and passes the string containing the file name. The effect is simulator dependent, but the 8080 simulator will attempt to load the file as an Intel hex format file and the 68000 simulator will attempt to load the file as a Motorola S-record file.

### 6.3.22 Sim-step

**Inputs**

None.

**Outputs**

None.

#### Description

Executes one instruction or step of the simulator.

### 6.3.23 Tape-Close

The following inputs are expected:

- A string representing the tape controller name.
- A string representing the output ("PUN" for punch) or input ("RDR" for reader).

#### Outputs

None

#### Description

Closes the file attached to the specified tape drive unit.

### 6.3.24 Tape-Open

The following inputs are expected:

- A string representing the tape controller name.
- A string representing the output ("PUN" for punch) or input ("RDR" for reader).
- A string representing the file name to attach to the tape drive.

#### Outputs

None

#### Description

Opens a file and attaches it to the specified tape drive unit.

### 6.3.25 Examples

The Lisp environment is useful for writing automated tests and for writing utilities to enhance the CLI.

#### Watch a Memory Location

The following Lisp program will execute instructions until the specified memory location changes.

```
(defun watch (addr)
  (setq old-value (meml addr))
  (dowhile (= old-value (meml addr))
    (sim-step)))
```

# Bibliography


[1] Intel. *8080/8085 Assembly Language Programming*. Intel, 1978.

[2] Motorola. *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*. Prentice-Hall, 4th edition, 1984.

[3] Motorola/NXP. *Motoroa M68000 Programmer's Reference Manual*. Motorola/NXP, 1992.

[4] MOS Technology. *MSC6502 Microcomputer Family Programming Manual*. MOS Technology, July 1976.

[5] Sean Young and Jan Wilmans. The undocumented z80 documented, 2005.

[6] Zilog/IXYS. *Z80 CPU User Manual*. Zilog, 2016.