Totem System Design

Jerry Lawrence jsl.totem@umlautllama.com

Fri May 17

Contributors
Patrick Stein pat@csh.rit.edu
Sean Graham grahams@csh.rit.edu

Contents

1 Overview

1.1 The Basic Concept

The Totem "*Tot*al *Em*ulation" design brings an object oriented approach to emulation of custom computer systems like arcade entertainment computers.

This emulator is composed of building blocks (CPU cores, sound cores, video display cores, memory cores, and so on.) which are shared libraries which can be lodaed and assembled as needed at runtime. They are assembled together to build an emulated computer architecture.

Unlike other emulators, the description on how they are assembled is not contained within the program itself, but rather in external XML recipe document which has a complete description of which modules are needed and how they interact with each other.

This XML recipe document, called a "Pole file" is read in by a central backbone, called the "Honcho". Once the document has been read in, the Honcho gathers all of the modules needed to build the hardware simulator as described in the document. These modules, or plugins are called "HEaD"s or "*H*euristic *E*mul*a*tion *D*evice"s.

The "Honcho" reads in the "Pole" to stack all of the "HEaD"s together appropriately. [1]

There are two approaches to construct this central backbone.

The first approach is a monolithic program which gets larger as more emulation core blocks are added. In this design, all of the emulation core blocks are contained within the program file. This approach is useful for systems where dynamic libraries are not as feasable.

The second approach is more flexible. All of the emulation core blocks are external to the central backbone, in shared libraries. If it was desired to only distribute the minimal system needed for "Bunny Blaster", you only need the driver file, the base backbone, the Z80 CPU core, the memory core, and the YF22A sound core. Or, conversely, if a better CPU core were available, you could just make that small download and update it, rather than downloading a behemothly large monolith containing all of the other modules which may not have changed.

This approach really helps out in a debugging application as well, since you can have two different memory cores. One which is made for speed, and one that has other things like visual GUI browsers, editors, and the like. If you wanted to debug the program you are running in the emulator, you would just load up the debug version of the memory core instead of the non-debug, and you can watch memory as it gets updated, or perhaps make your own tweaks to see how gameplay changes.

A third advantage of this approach is that you could have two sets of "drivers" for a game. The first set is as close to accurate as can be. It will emulate ALL of the hardware as close to the physical hardware as is possible.

[1] Okay, so maybe it's not the best analogy, trying to corrolate an Emulator with a Totem Pole...

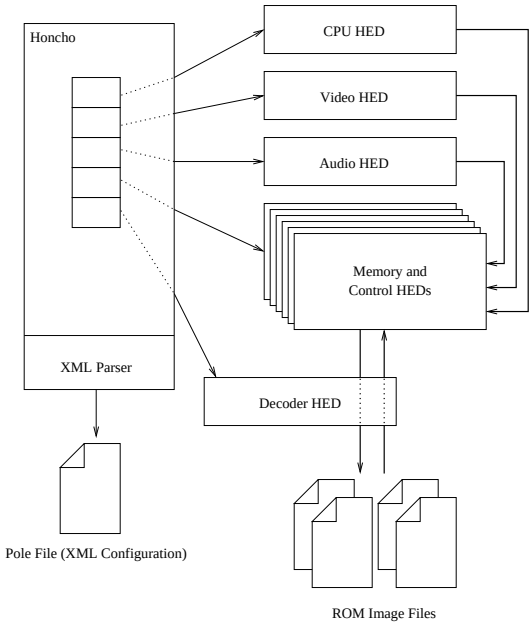

Figure 1: An example emulation topology.

There can also be a second set which is designed to be a little more loose, made simply for playing the game. This second set might use less memory, less processor time and so on. Since it's not trying to emulate the hardware perfectly, there is a lot that can be overlooked, since some of the hardware might not ever be referenced.

It also opens up the game for other independant advances. For example, you could replace the Pac-Man video hardware with your own design that allows for sprites of larger than 16 pixels square, with more than four colors. A programmer could make a high-resolution Pac-Man [2] easily by just switching the standard video driver with their own.

An example topology graph for a basic emulator is shown in Figure 1.1.

1.2 The Rest of This Document

The rest of this document will go into detail on how a preliminary version of this emulator will be constructed. Due to unforseen design and implementation issues, the final design will probably differ from the base design as explained in the pages to follow.

[2] Ref: Namco Classics Volume 2's "Pac-Man Arrangement" which uses the original game play but with updated graphics.

2 Reference Work

This section will describe a few of the commonly available emulators, describe some of their advantages and faults. Also to be described is which elements from these emulators will be utilized for Totem.

2.1 MAME

URL: `http://www.mame.net`
Author: Nicola Salmoria et al.

2.1.1 History

From the Mame Homepage:

On December 24th, 1996, Nicola Salmoria began working on his single game emulators (for example Multi-Pac), which he merged into one program during January 1997. He named the accomplishment by the name of Multiple Arcade Machine Emulator, or MAME for short (pronounced as the word 'maim' in English, other languages may differ). The first official release was MAME 0.1, which was released on the evening of February 5th, 1997 (23:32 +0100). Using a modular and portable driver oriented architecture with an open source philosophy, it soon grew into immense proportions.

The current version supports well over 3000 ROM sets, more than 1700 unique games. Because MAME releases happen whenever they are ready, at one point the wait between new versions was almost 4 months. To help the agony of the users, a public beta system is now used, with a beta release happening every 2-3 weeks on an average. Also a work-in-progress-page exists, if you really want to know the latest information.

Even though MAME allows people to enjoy the long-lost arcade games and even some newer ones, the main purpose of the project is to document the hardware (and software) of the arcade games. There are already many dead arcade boards, whose function has been brought to life in MAME. Being able to play the games is just a nice side-effect. The huge success of MAME would not be possible without the talent of the programmers who joined to form the MAME team. At this moment, there are about 100 people on the team, but there is a large number of contributors outside the team too. Nicola Salmoria is still the coordinator of the project.

2.1.2 Advantages

- Very portable - MAME has been ported to many different operating systems and computer architectures including unexpected devices such as digital cameras.
- Extensive game coverage - With over 3000 games currently supported, the breadth of the types of computer hardware emulated by MAME is very wide.

- Good documentation of the hardware – There are many aspects of the architectures of each of the pieces of hardware emulated that are documeted enough to get the game working.

2.1.3 Faults

- Monolithic - MAME is only available in one larger-than-bite-sized chunk.
- Immense - As they tout themselves, MAME is over a million lines of code right now. A lot of that is hardware emulation cores, but quite a bit of it is game drivers, structures of data in the program.
- Multiple libraries required to build it - Instead of sticking with just one game programming library (like Allegro or libSDL), it requires two which have overlapping functionality. (Allegro and SEAL.) Allegro could have easily supported the functionality that SEAL provides. They also use another assembler, NASM.
- Hardware behavior not documented very well – They document that the machines have certian bits in their dipswitch settings, but not what they do, or how they do it. Many arcade machines use custom hardware chips which perform graphics operations, but there's no way to determine how the game would perform if one of these chips were to become faulty. The overall behavior of the machine is documented, not the behavior of the sub-parts.
- Driver files are entirely within the source code – If there is a new variant of a game that a user wants to add into MAME, they must either rename the rom files to be what is expected, losing any specificness of this new romset, or they must modify the source code, and recompile the emulator. Not every end user has the means or knowledge to do the latter, which is the better response for still documenting the game.

2.2 Sparcade

URL: `http://www.sparcade.freeserve.co.uk/`
Author: David Spicer

2.2.1 History

Sparcade started at about the same time as MAME. Sparcade is closed source, only developed by one person, for one system, and thus it can never be as extensive as MAME is. It did approach the emulation idea in a different way, and has different performance than MAME.

2.2.2 Advantages

- CPU cores are in assembler - Since Sparcade is made specifically for one target architecture, it has been tweaked to run very fast on that type of machine. One of these tweaks is that the cpu emulation cores have been written in machine language.
- Games are somewhat defined in .RAT files - To add a new variant, all that needs to be done is modification or creation of these .RAT data files to look for the new rom files. Recompiling is not necessary.
- Multiple names for rom filenames (See the "Tron" .RAT file for an example.) - This lets the emulator look for different versions of a rom file by name, until it finds one. For example, "`boot0`", "`pacman0`", "`pacman.6e`", might all be the same rom for a specific game.
- Most games retain their states between sessions - This means that if you were to close down a game, and start it up later on, it will be exactly where you left it. This also means that high scores are all saved with no extra code needed.

2.2.3 Faults

- CPU cores are in assembler - This means that the heart of this emulator, is specific to one architecture, Intel x86 machines running a MS-DOS variant.
- Non portable - MS-DOS / Windows only - Since this is a closed source project, there's no chance of it being ported to other architectures, not that you could, since the main CPU cores are written in assembler anyway.
- Source not available - There's no way for another person to learn from this project. People can only use it and be happy with what they have.

3 Design Goals

As would be expected, the design for Totem takes into account some of the design decisions and architectures of other emulators. The design for Totem is based on the good ideas of these emulators, as well as some new ideas not yet implemented in an emulator architecture.

Totem will be upgradable and user-changable. Totem will have a design that will enable different versions of the core modules to be swapped in and out at will. This will allow the user to use either debug, accurate, fast or experimental versions of the core modues, at their own discretion.

Totem will be portable. It should require little to no effort to bring the emulator over to a new operating system or computer architecture. This also means that no technologies that are designed for a specific architecture or operating system should be used.

Totem will be open-sourced. It will enable multiple people to work on it, each person providing different aspects of code, different approaches, different ideas and so on, to produce a better end product.

Totem will be extensible. It will be easy to add updated or new emulation cores and supported games to the emulator. It will be flexible enough to enable emulation of arcade hardware, computer architectures and other computer-based simulation systems.

Totem will have an external configuration document system, enabling the end user to create their own drivers, or experiment with configurations of working drivers without needing to recompile the emulator. This document should also be used to describe the hardware in normal plain text within the document's structure, rather than a proprietary binary format.

4 Technologies

The portability of the system will be provided by writing it in ANSI C++ which is available for practically every platform. Totem will also use the SDL [3] (Simpe DirectMedia Layer) library, which is available for many platforms, including Macintosh, PC, Unix, and BeOS.

The modularity of the system will be provided by having each of the core modules be in their own shared library, which will be linked and loaded at runtime as needed.

The external configuration documents will be in XML [4]. (eXtensible Markup Language.) The DTD (Data Type Description) describing how this file is layed out can be found later in this document in secion 7.

[3] http://www.libsdl.org
[4] http://www.w3.org/XML/

5 The Honcho

The Honcho is the main backbone of the emulator. This is the primary executable, application, program, whatever you want to call it. The Honcho is responsible for all of the central nervous system like aspects of the emulator.

The basic tasks of the Honcho, as the emulator runs are as follows:

1. Find and parse in the pole file.
2. Find all required HEaD modules.
3. Initialize all HEaD modules.
4. Start the CPUs
5. Wait for the user to shut down the system.
6. Shutdown the HEaD modules.
7. Free up memory and exit.

Which basically works down into three main categories:

5.1 XML Parser

The Honcho parses the configuration file (the 'pole') for the architecture to be emulated. The pole file is an XML document which descirbes everything about the hardware to be emulated, which emulation modules and external files are required, and how they are all connected to each other.

The DTD for the pole file is in section 7.1, and a sample file is in section 7.2.

The Honcho will create lists of the modules it knows about, and various other information about them from this XML document.

5.2 Linker

Once all of the needed objects have been found, the Honcho will link them all in, and initialize them. This will use a standard shared library linking mechanism (exact design is TBD).

Once they are all linked in, the Honcho will call all of the modules to initialize them. The Honcho will then manually connect any modules that need to be told with whom they connect. This is only necessary for some modules that dont inherently know what they need.

After the main linking operation has completed, the Honcho will call all of the objects again, but this time through their "`connect()`" method. This is when objects will connect to eachother. They will query the Honcho for references to other objects that it should know about.

5.3 Traffic Cop

Once runtime has set in, the Honcho just sits back and lets all of the emulation modules do their respective things. The Honcho will wait at this point for a shutdown message to come through, at which point it will stop, shutdown and disconnect all of the modules, and finally, exit.

6 Emulation Core HEaD Modules

A HEaD, (Heuristic Emulation Device) is the name given to the building blocks of the emulator. Computer architectures can be emulated simply by arranging and connecting together these HEaDs in different ways.

6.1 Base HEaD Class

All of the following HEaD module API's will inherit from this class. This takes care of the basic common HEaD functionalities.

These break down into the two setup phases and the shutdown phase of runtime. This is explained in greater detail in the Lifecycle section (section 8) below.

All HEaDs must have a unique name. This is the name as defined for this module in the pole file.

12 ⟨*Base HEaD API* 12⟩≡

```
void initialize( char * baseHEaD_name );
void connect( void );
void shutdown( void );
```

6.2 CPU HEaD

Each CPU that is being emulated will be in its own CPU HEaD core module. For each instance of each CPU, another instantiation of the HEaD module will be created.

Each CPU HEaD stores info about its connections to memory, its speed, and other such information. This information is told to it by the Honcho at the second phase of initialization. The CPU HEaD will run on its own, independently of the other HEaD modules. It will initialize the appropriate timer routines at setup time.

The CPU cores will probably consist of a well known, proven codebase, with this API grafted onto it. The only modifications necessary to this code base is that it must be modified to use the Memory HEaD API calls for interacting with memory. Rather than the memory being local to the CPU module, it is stored externally. More about this in the Memory section below.

The CPU module can have memory modules attached to it. The size and length of the memory for each block attached is gleaned from the Memory HEaD which the CPU HEaD gets attached to. All that needs to be set is whether it is readable, writable, or both.

This information is stored in two sorted lists inside the CPU HEaD module. One list is for writable memory, the other is for readable memory. If the block is both readable and writable, it will be added to both lists.

If multiple memory blocks are added that have the same start address, the last one added will be the one that is used. If overlapping memory blocks are added, the one at the higher start address will be the one that is accessed.

13a ⟨*CPU HEaD API* 13a⟩≡ 13b▷

```
#define CPU_FLAGS_MEM_READ      0x01
#define CPU_FLAGS_MEM_WRITE     0x02

attach_memory(
        char * memoryHEaD_name,
        unsigned long start_address,
        int flags
);
```

Also needed is a way to attach ports to the CPU. This is stored in a third list in the module. These are handled the same way as memory, only that they will probably have a different result behavior

13b ⟨*CPU HEaD API* 13a⟩+≡ ◁13a 14a▷

```
connect_port(
        char * porthandler_name,
        int     portnumber
);
```

The user may wish to pause or change the speed of the processor, to simulate a bad clock, or to slow down the game.

14a ⟨*CPU HEaD API* 13a⟩+≡ ◁13b 14b▷
```
set_speed(
        float mhz
);

pause();
```

Due to the nature of CPUs, there's really very little interaction that external devices can do to it. Primarilly, it's just an interrupt and a reset.

14b ⟨*CPU HEaD API* 13a⟩+≡ ◁14a 14c▷
```
reset();
```

14c ⟨*CPU HEaD API* 13a⟩+≡ ◁14b
```
interrupt();
```

6.3 Memory HEaD

The Memory module is probably the most complicated core that we will need.

The memory HEaD does not know where in the processor's space it sits, as the same memory HEaD can be accessed by two different processors. This means that the offset for the memory accesses will need to be subtracted by the CPU HEaD before the memory can be accessed through the API.

In order to act like the real hardware, if the address is not found, a zero, random numnber, or repeating memory (TBD) will be returned instead of real memory.

14d ⟨*Memory HEaD API* 14d⟩≡ 14e▷
```
char * memoryHEaD_name;
unsigned long length;
char * data;

void clear( void );
```

14e ⟨*Memory HEaD API* 14d⟩+≡ ◁14d 14f▷
```
load_from_file( char * filename, unsigned long baseaddress );
save_to_file( char * filename, unsigned long baseaddress );
```

14f ⟨*Memory HEaD API* 14d⟩+≡ ◁14e 15a▷
```
char      read_8(  unsigned long address );
int       read_16( unsigned long address );
long      read_32( unsigned long address );
long long read_64( unsigned long address );
```

15a ⟨*Memory HEaD API* 14d⟩+≡ ◁14f

```
void write_8(  unsigned long address, char data );
void write_16( unsigned long address, int data );
void write_32( unsigned long address, long data );
void write_64( unsigned long address, long long data );
```

6.4 Sound HEaD

Due to differing game types, there will be two basic sets of functions in the Sound API. The first are for emulated analog games, the second are for regular emulated hardware.

The design for this module so far seems extremely flawed and will require more research to determine the proper API.

For some emulated hardware, they will look at registers in memory as defined by the initialization calls.

These initialization calles are yet to be defined.

This following call is for games that require "emulation" of analog sound circuits. This will be provided initially as a wave file player. Modules of this type can be re-used. This should only need to be written once.

15b ⟨*Sound HEaD API* 15b⟩≡

```
void play_sound( unsigned long soundnumber );
```

If there were an architecture that used PC ISA sound cards, it might implement the sound interface HEaD as a Porthandler HEaD instead of a Sound HEaD.

6.5 Video HEaD

For games that require different chunks of ram (character, color, sprite, etc) they will query the Honcho to retrieve the modules that it needs. This is covered in the lifecycle section in more depth.

The video HEaD is responsible for opening a display window or screen, as well as communicating with it.

At connection startup time, the video HEaD will retrieve all of the memory blocks through the Honcho that it needs to render the graphics.

The Video HEaD might have a control connection that it requests as well which might be how the user chan change framerate and such.

Issue 1 *There are some things that need to be synced between the video HEaD and the CPU HEaD. (Video sync for example) This method must be designed.... Perhaps there should be a timer HEaD as well to sync modules together?*

6.6 Porthandler HEaD

The port handler HEaD is basically just a callback that handles any port IO for a processor.

The Porthandler HEaD is a derived class of the Memory HEaD module.

The only main difference is that instead of requesting memory from a specific address, the CPU module will be requesting data from a specific port.

Generally, this will be connected to a specific port number and range on the CPU module. From within the Porthandler HEaD, the ports will start at `0`, commonly called the "base port" for the device, and go for as many ports as necessary for the device.

Although if something like parallel port emulation were to be implemented, it might be hooked up at port `0x378`, and be `3` ports long. Then, the Porthandler HEaD would get requests for data at ports `0x00`, `0x01`, and `0x02`, which are the data, status and control ports of a standard parallel port. The HEaD architecture will enable two of these to be connected to the CPU module for example at ports `0x378` and `0x278` to simulate a computer with two parallel ports.

6.7 Control HEaD

In mose cases, this is a derived class of the Memory HEaD module. This will usually consist of one byte or so of ram space which is accessed as if it were regular ram, but the other side of it will generally be something within the emulator itself; whether it be the joystick interface or a dipswitch menu system.

The Pole file will tell the control modules how the bits that they are responsible for are connected to the real world.

7 Pole File

The Pole is the data file that describes how the HEDs are arranged. It will list how large each memory block is, the speed of the cpu, which graphics rendering method to use and so on.

7.1 Totem Pole DTD

Issue 2 *The data decode in here works great for when a single ROM needs to be loaded into a memory block, but what if we have one ROM that needs to get decoded into multiple memory blocks, or multiple ROMS that need to be decoded into a single memory block? For example; Ms.Pac-Man AUX board, where one ROM overlays 4 rom spaces.*

17 ⟨*totempole.dtd* 17⟩≡

```
<?xml version="1.0" standalone="yes"?>

<!-- DTD for a Totem Pole file -->

<!-- NOTE: this is incomplete.  It is missing other control inputs, etc. -->

<!ELEMENT TotemPole (GameInformation, DriverInformation, Module+)>

<!-- GameInformation elements -->
<!ELEMENT Name          (#PCDATA)>
<!ELEMENT Year          (#PCDATA)>
<!ELEMENT Copyright     (#PCDATA)>
<!ELEMENT ROMName       (#PCDATA)>
<!ELEMENT Inherits      (#PCDATA)>

<!ELEMENT GameInformation (
        Name,
        Year,
        Copyright+,
        ROMName,
        Inherits*
)>


<!-- DriverInformation elements -->
<!ELEMENT Version       (#PCDATA)>
<!ELEMENT Author        (#PCDATA)>
<!ELEMENT EMail         (#PCDATA)>
<!ELEMENT Thanks        (#PCDATA)>

<!ELEMENT DriverInformation (
```

```
        Version,
        Author+,
        EMail+,
        Thanks*
)>


<!-- ------------------------------------------------------------ -->
<!-- Module information elements -->
<!ELEMENT Class         (#PCDATA)>
<!ELEMENT Library       (#PCDATA)>
<!ELEMENT InternalName  (#PCDATA)>

<!ELEMENT Module (
        Class,
        Library,
        InternalName,
        (
            Memory      |
            DataDecode  |
            Video       |
            CPU         |
            Control
        )
)>

<!-- ------------------------------------------------------------ -->

<!-- Memory Module -->
<!ELEMENT Memory        (Datasize, Initialize)>
<!ELEMENT Datasize      (#PCDATA)>
<!ELEMENT Initialize    ( Random | EncodedFile+ | File )>
<!ELEMENT EncodedFile   (File+, Decoder)>
<!ELEMENT File          (#PCDATA)>
<!ELEMENT Decoder       (#PCDATA)>
<!ELEMENT Random        EMPTY>

<!-- ------------------------------------------------------------ -->

<!-- DataDecode Module -->
<!ELEMENT DataDecode    EMPTY>

<!-- ------------------------------------------------------------ -->

<!-- Control Module -->
<!ELEMENT Control       (Bit+)>
```

```
<!ELEMENT Bit           (Value, (Joystick | Test | Coin | Start))>
<!ELEMENT Value         (#PCDATA)>

<!-- Joystick input device -->
<!ELEMENT Joystick      (Port, (Up | Down | Left | Right | Fire)>
<!ELEMENT Port          (#PCDATA)>
<!ELEMENT Up            EMPTY)>
<!ELEMENT Down          EMPTY)>
<!ELEMENT Left          EMPTY)>
<!ELEMENT Right         EMPTY)>
<!ELEMENT Fire          (#PCDATA)>

<!-- Test Switch -->
<!ELEMENT Test          EMPTY)>

<!-- Coin Switch -->
<!ELEMENT Coin          (#PCDATA))>

<!-- Start Button -->
<!ELEMENT Start         (#PCDATA))>

<!-- ------------------------------------------------------------ -->

<!-- CPU Module -->
<!ELEMENT CPU (
        Speed,
        CPUMemory+
)

<!ELEMENT Speed         (#PCDATA)>

<!ELEMENT CPUMemory (
        Read?,
        Write?,
        Base,
        Size,
        InternalName
)>

<!ELEMENT Read          EMPTY>
<!ELEMENT Write         EMPTY>
<!ELEMENT Base          (#PCDATA)>
<!ELEMENT Size          (#PCDATA)>

<!-- ------------------------------------------------------------ -->
```

```
<!-- Video Module -->
<!ELEMENT Video (
        Screen,
        VideoMemory
)>

<!ELEMENT Screen (
        Ratio,
        Resolution
)>
<!ELEMENT Ratio         (#PCDATA)>
<!ELEMENT Resolution    (#PCDATA)>

<!ELEMENT VideoMemory (
        Handle,
        InternalName
)>
<!ELEMENT Handle        (#PCDATA)>
```

7.2 Sample File

Here is a sample Pole file:

21 ⟨*pacmanm.pol* 21⟩≡

```
<!DOCTYPE TotemPole SYSTEM "TotemPole.dtd">

<!-- NOTE: this pole file is incomplete. -->

<TotemPole>
    <!-- basic game information -->
    <GameInformation>
        <Name> Pac-Man (Midway License)</Name>
        <Year> 1981 </Year>
        <Copyright> 1980 Namco, Midway </Copyright>

        <ROMName> pacmanm </ROMName>
        <Inherits> pacman </Inherits>
    </GameInformation>

    <DriverInformation>
        <Version> 0.01 </Version>
        <Author> Jerry Lawrence </Author>
        <EMail> jsl.totem@absynth.com </EMail>
        <Thanks> www.mame.net </Thanks>
        <Thanks> Nicola Salmoria </Thanks>
    </DriverInformation

    <!-- some sort of memory file decoder for the rom space -->
    <Module>
        <Class> DataDecode </Class>
        <Library> my_pacman_decoder </Library>
        <InternalName> pac_data_decoder </InternalName>
        <DataDecode/>
    </Module>

    <!-- specify 2kb of ram, randomly initialized -->
    <Module>
        <Class> Memory </Class>
        <Library> ram_module </Library>
        <InternalName> pac_background_ram_01 </InternalName>
        <Memory>
            <Datasize> 0x1000 </Datasize>
            <Initialize> <Random/> </Initialize>
        </Memory>
    </Module>
```

```
<!-- specify 2kb of rom, preloaded with a rom file -->
<Module>
    <Class> Memory </Class>
    <Library> rom_module </Library>
    <InternalName> pac_rom_0000 </InternalName>
    <Memory>
        <Datasize> 0x1000 </Datasize>
        <Initialize> <File> pacman.6e <File> </Initialize>
    </Memory>
</Module>
<Module>
    <Class> Memory </Class>
    <Library> rom_module </Library>
    <InternalName> pac_rom_1000 </InternalName>
    <Memory>
        <Datasize> 0x1000 </Datasize>
        <Initialize>
            <EncodedFile>
                    <File> pacman.6f <File>
                    <Decoder> pac_data_decoder </Decoder>
            </Initialize>
    </Memory>
</Module>

...

<!-- and the control input -->
<Module>
    <Class> Control </Class>
    <Library> PacMan </Library>
    <InternalName> pac_IN0 </InternalName>
    <Control>
        <Bit>
            <Value>0x01</Value>
            <Joystick> <Port>0</Port> <Up/> </Joystick>
        </Bit>
        <Bit>
            <Value>0x02</Value>
            <Joystick> <Port>0</Port> <Left/> </Joystick>
        </Bit>
        <Bit>
            <Value>0x04</Value>
            <Joystick> <Port>0</Port> <Right/> </Joystick>
        </Bit>
        <Bit>
            <Value>0x08</Value>
```

```
            <Joystick> <Port>0</Port> <Down/> </Joystick>
        </Bit>
        <Bit>
            <Value>0x10</Value>
            <Test/>
        </Bit>
        <Bit>
            <Value>0x20</Value>
            <Coin> 0 </Coin>
        </Bit>
        <Bit>
            <Value>0x40</Value>
            <Coin> 2 </Coin>
        </Bit>
        <Bit>
            <Value>0x80</Value>
            <Coin> 3 </Coin>
        </Bit>
    </Control>
</Module>

<!-- the CPU gets initialized once all of its RAM/ROM has been loaded -->
<Module>
    <Class> CPU </Class>
    <Library> Z80_core </Library>
    <InternalName> cpu_01 </InternalName>
    <CPU>
        <Speed> 3.072000 </Speed>
        <CPUMemory>
            </Read>
            <Base> 0x0000 </Base>
            <Size> 0x1000 </Size>
            <InternalName> pac_rom_0000 </InternalName>
        <CPUMemory>
        <CPUMemory>
            </Read>
            <Base> 0x1000 </Base>
            <Size> 0x1000 </Size>
            <InternalName> pac_rom_1000 </InternalName>
        <CPUMemory>
        <CPUMemory>
            </Read>
            <Base> 0x2000 </Base>
            <Size> 0x1000 </Size>
            <InternalName> pac_rom_2000 </InternalName>
        <CPUMemory>
```

```
            <CPUMemory>
                </Read>
                <Base> 0x3000 </Base>
                <Size> 0x1000 </Size>
                <InternalName> pac_rom_3000 </InternalName>
            <CPUMemory>
            <CPUMemory>
                </Read> </Write>
                <Base> 0x4000 </Base>
                <Size> 0x0400 </Size>
                <InternalName> pac_background_ram_01 </InternalName>
            <CPUMemory>
            ...
            <CPUMemory>
                </Read>
                <Base> 0x5000 </Base>
                <Size> 0x0001 </Size>
                <InternalName> pac_IN0 </InternalName>
            </CPUMemory>
        </CPU>
    </Module>

    <!-- here is where the video hardware is defined -->
    <Module>
        <Class> Video </Class>
        <Library> PacMan </Library>
        <InternalName> pac_video </InternalName>
        <Video>
            <Screen>
                <Ratio> 28.27 </Ratio>
                <Resolution> 288x224 </Resolution>
            </Screen>
            <!-- these slices are defined in the module itself -->
            <VideoMemory>
                <Handle> Background_RAM </Handle>
                <InternalName> pac_background_ram_01 </InternalName>
            </VideoMemory>
            <VideoMemory>
                <Handle> Color_RAM </Handle>
                <InternalName> pac_color_ram_01 </InternalName>
            </VideoMemory>
            <VideoMemory>
                <Handle> Sprite_RAM </Handle>
                <InternalName> pac_Sprite_ram_01 </InternalName>
            </VideoMemory>
            ...
```

```
        </Video>
    </Module>
<TotemPole>
```

8 Emulation Lifecycle

8.1 Setup - Pole Parsing

The first thing that happens is that the Honcho parses the configuration file (the 'pole') for the architecture to be emulated.

The HEaD modules (Heuristic Emulation Device) are each of the core emulation modules, the building blocks, which can be connected to build a complete emulation of a computer architecture.

8.2 Setup pass 1 - Initialization and Binding

At this point, the HEaD modules will be found and initialized through their "`initialize()`" method.

When a HEaD module is initialized by the Honcho, it is given a unique name. This name is the "Internal Name", as specified in the pole file. It is by this name that the modules can reference each other.

The HEaD modules are registered with the Honcho at this time. That is to say that the Honcho remembers all of the modules that it has initialized.

8.3 Setup pass 2 - Manual Connection

The Honcho will now call some of the initialized modules to tell them where they should link to. This is only necessary for some modules that don't inherently know what they need.

Some connections are described in the pole file. This procedure is for the Honcho to tell the objects involved in those connections about eachother. The CPU module is this way, for example.

Since the CPU module is generic, it does not know what the configuration of the ram and rom is that it needs. It does not know where ram and rom sit in its memory space. The Honcho has already parsed the pole file and knows what needs to be connected where.

For example, the Honcho will tell the CPU that there is a memory block called "`rom_0000`" at memory address `0x0000`.

8.4 Setup pass 3 - Self-Connection

The third pass of setup is for telling the modules to connect to the things that they know that they need. The Honcho will call all of the modules in the same order as described above. The Honcho will tell these modules that it is okay to request each other at this time. It will do this by calling their "`connect()`" method.

All modules will be called in the order that they are listed within the pole file.

The modules will then call the Honcho to find the modules that it needs. For example, if there is a video module that needs access to a chunk of memory, the video module will contact the Honcho, stating who it is, and will ask for a name that it knows about that corresponds to a name in the configuration file.

The interchange may go something like this:

Video module: "Hi. I'm the video module that you named `foobar`. I need access to a chunk of memory that I call `COLOR_RAM`. What do you call it?"

Honcho: "I call the `COLOR_RAM` that you know about '`_color_ram_00`'."

Video module: "I need a reference to `_color_ram_00`."

Honcho: "Here you go!" ... and the Honcho returns the reference.

For example, the Namco/Pac-Man video driver will request 7 memory banks: "`Background_RAM`", "`Color_RAM`", "`Sprite_RAM`", "`Palette_ROM`", "`Color_ROM`", "`Character_ROM`", and "`Sprite_ROM`". There should be entries in the pole file that define these blocks, so that the skeleton can have them ready. These bank names are hardcoded within the HEaD module itself.

The skeleton will know which HEaD module is calling it, so it will know which requested HEaD module it is corrolated with. This will enable a configuration where we have multiple video driver modules (think of a multiple screen Pac-Man) that both request a HEaD called "`Background_RAM`", but get different RAM chunks.

This also will let the video driver HEaD module to be generic. The video HEaD module only knows that it needs a chunk of ram called "`Background_RAM`", and not where it is in the memory space of the CPU or anything like that.

8.5 Runtime

At this point, everything is basically controlled by the CPU modules that have been loaded in. They will access and modify the ram in the RAM modules, and so on. They will retrieve control information from the Control modules, and basically do their thing.

Although the emulator will spend most of its time doing this task, there really isn't much to write here. The emulator's modules will basically *do their thing*[5] until the user shuts down the emulator, or an illegal instruction happens on the CPU.

[5] Copyright 2001. All rights reserved.

8.6 Shutdown

The first thing that will happen is that the HEaD modules will be called through their shutdown method. The modules will be called in the reverse order that they were initialized. They are responsible for freeing up any memory, saving data files, and so on, as needed.

The modules at this point may decide to save all of their state information relating to the game loaded in, so that the exact state of the emulated computer can be restored when this emulation configuration is run again in the future.

A Totem Glossary

Arcade Game A machine with embedded, usually custom hardware, designed to absorb your quarters.

Totem Totem is an abbreviation for Total Emulation.

Pole The data recipe file which describes how all of the HEaDs are arranged and defined. It describes how the elements of an emulation solution are related to each other. Examples would be a pole file for Pac-Man hardware, another for Pengo hardware, and another for Apple II hardware.

HEaD HEaD is an abbreviation for *H*euristic *E*mul*a*tion *D*evice. A HEaD is one emulation core module. Some examples are the Z80 core, the Memory core, and a FM sound synthesizer core.

Honcho The Honcho is the main central nervous system of the emulator. It is basically just a mechanism that combines an XML parser to read in the pole file, and a mechanism to load and connect the various needed HEaD modules. It also enables the HED modules to request eachother for communications purposes.