0.1 About the Source Code

Commander Keen series 1-3 and 4-6 source code is not available as the current owner Zenimax **??** has, as of writing this book, no interest in selling intellectual properties. Luckily the ownership of Commander Keen: Keen Dreams was in the hands of Softdisk. In June 2013, developer Super Fighter Team licensed the game from Flat Rock Software, the then-owners of Softdisk, and released a version for Android devices.
The following September, an Indiegogo crowdfunding campaign was started to attempt to buy the rights from Flat Rock for US$1500 in order to release the source code to the game and start publishing it on multiple platforms. The campaign did not reach the goal, but it's creator Javier Chavez made up the difference, and the source code was released under GNU GPL-2.0-or-later soon after.

0.2 Getting the Source Code

The source code is made available via `github.com`. It is important to take the the source code for the shareware version 1.13, otherwise you run into issues due to incompatible map headers. To get the correct source code

```
$ git clone https://github.com/keendreams/keen.git
$ cd keen
$ git checkout a7591c4af15c479d8d1c0be5ce1d49940554157c
```

0.3 First Contact

Once downloaded via `github` a folder 'keen' is created with all source files inside. `cloc.pl` is a tool which looks at every file in a folder and gathers statistics about source code. It helps for getting an idea of what to expect.

```
$ cloc keen

52 text files.
52 unique files.
7 files ignored.

-----------------------------------------------------------
Language          files        blank      comment        code
-----------------------------------------------------------
C                    20         4008         5361       14893
Assembly              5          992         1114        2688
C/C++ Header         19          508          665        1603
Markdown              1           18            0          40
DOS Batch             1            0            0          13
-----------------------------------------------------------
SUM:                 46         5526         7140       19237
-----------------------------------------------------------
```

The code is 85% in C with assembly[1] for bottleneck optimizations and low-level I/O such as video or audio.

Source lines of code (SLOC) is not a meaningful metric against a single codebase but excels when it comes to extracting proportions. Commander Keen with its 19,237 SLOC is very small compared to most software. `curl` (a command-line tool to download url content) is 154,134 SLOC. Google's Chrome browser is 1,700,000 SLOC. Linux kernel is 15,000,000 SLOC.

[1] All the assembly in Keen is done with TASM (a.k.a Turbo Assembler by Borland). It uses Intel notation where the destination is before the source: `instr dest source`.

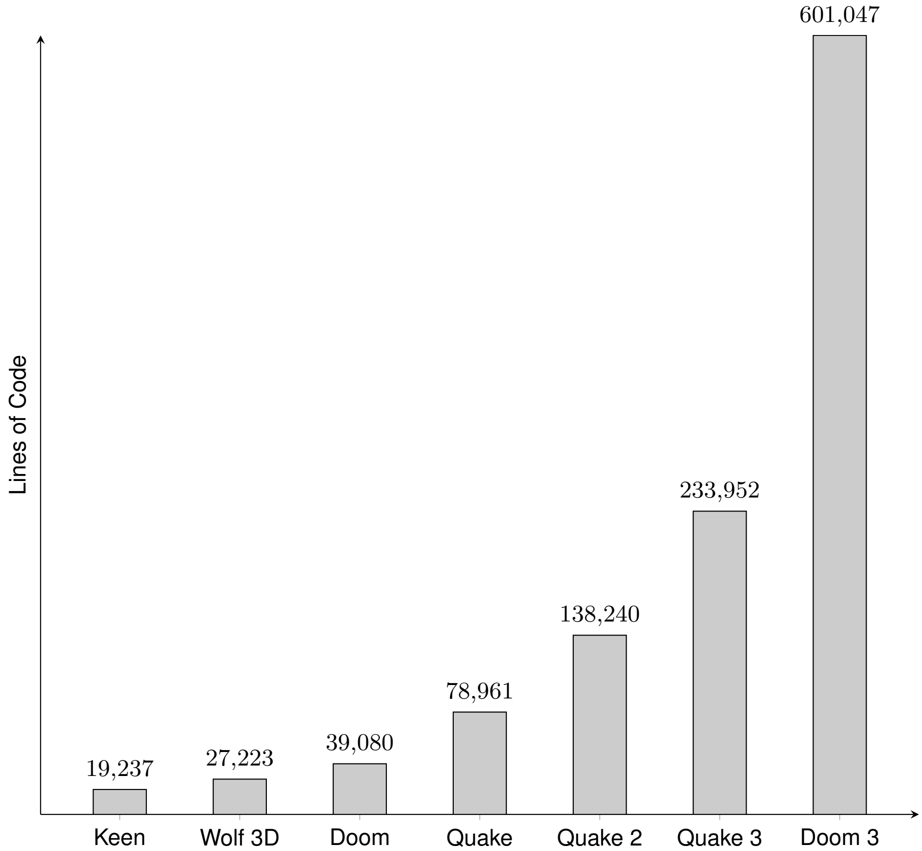

Figure 1: Lines of code from id Software game engines.

The archive contains more than just source code; it also features:

- `static` folder: Static object files (will be explained later).
- `lscr` folder: Load and decompress lzw data files.
- `README:` How to build the executable.

0.4 Compile source code

Now let's start to compile the source code. To compile the code like it's 1990 you need the following software:

- Commander Keen source code.

- DosBox.
- The Compiler Borland C++ 3.1.
- Commander Keen: Keen Dreams 1.13 shareware (for the assets).

After setting up the DosBox environment, with Borland C++ 3.1 installed (You can find a complete tutorial in "Let's compile like it's 1992" on `fabiensanglard.net`) download the source code via github.
Once you start DosBox and change directory to the `keen` folder, first create the folder where we create our compiled object files.

```
mkdir OBJ
```

Then we need to create the static `OBJ` files.

```
chdir STATIC
make.bat
```

Once the static object files are created move back to the `keen` folder and open Borland C++. Open the `kdreams.prj` project file. Before we can start compiling we need to set the correct directories. Select Options -> Directories and change the values as follow:

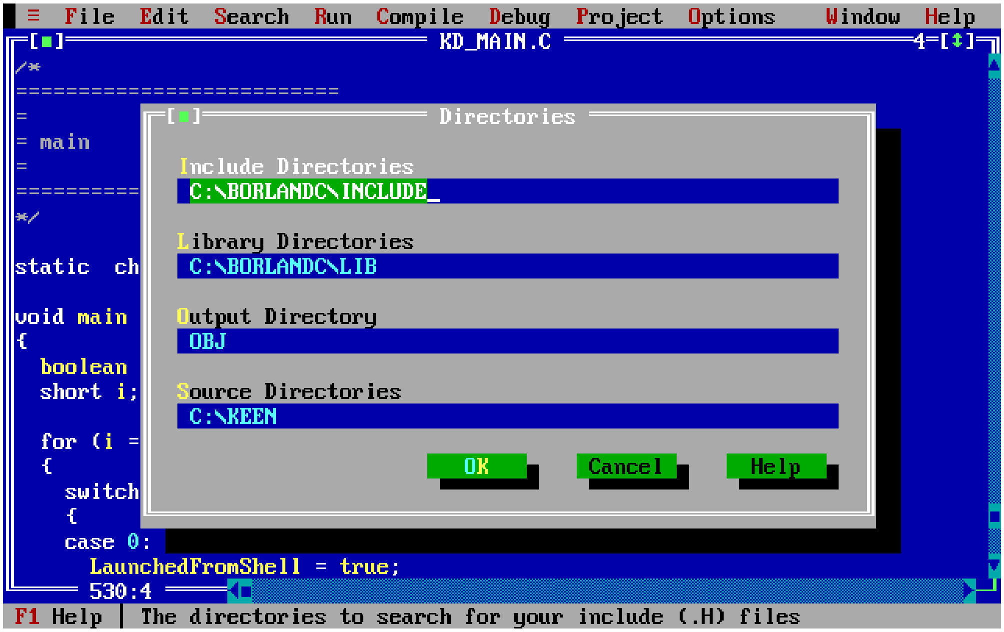

Figure 2: Borland C++ 3.1 directory settings

Now it's time to compile. Go to Compile -> Build all, and voila! The final step is to copy `kdreams.exe` to the Keen shareware folder. Now you can play your compiled version of Commander Keen.

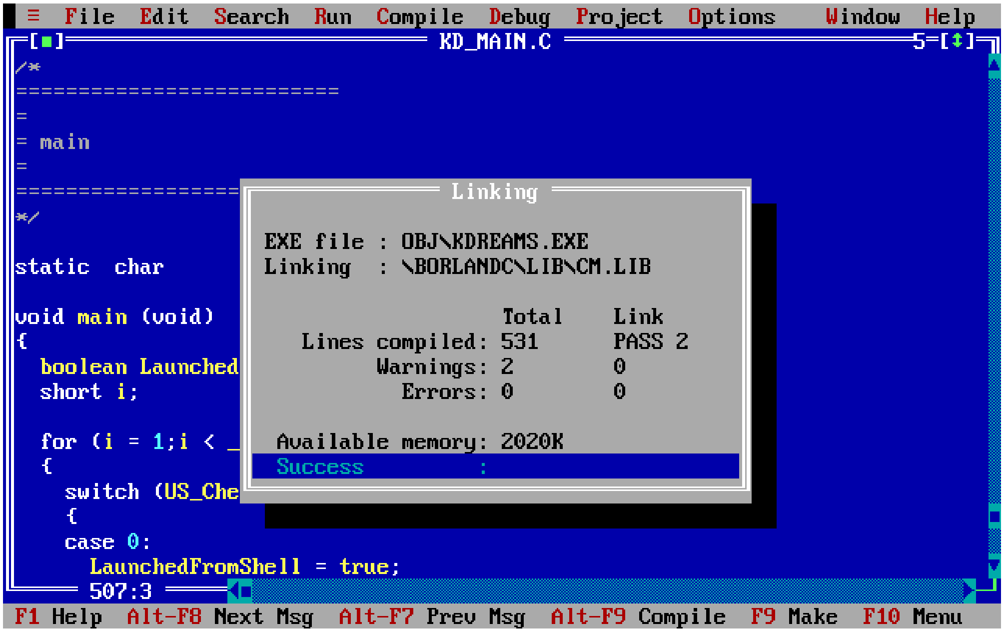

Figure 3: Commander Keen compiling

0.5 Big Picture

The game engine is divided in three blocks:

- Menu engine which lets users configure the game.
- 2D game renderer where the users spend most of their time.
- Sound system which runs concurrently with either the Menu or 2D renderer.

The three systems communicate via shared memory. The renderer writes music and sound requests to the RAM (also making sure the assets are ready). These requests are read by the sound "loop". The sound system also writes to the RAM for the renderers since it is in charge of the heartbeat of the whole engine. The renderers update the world according to the wall-time tracked by `TimeCount` variable.

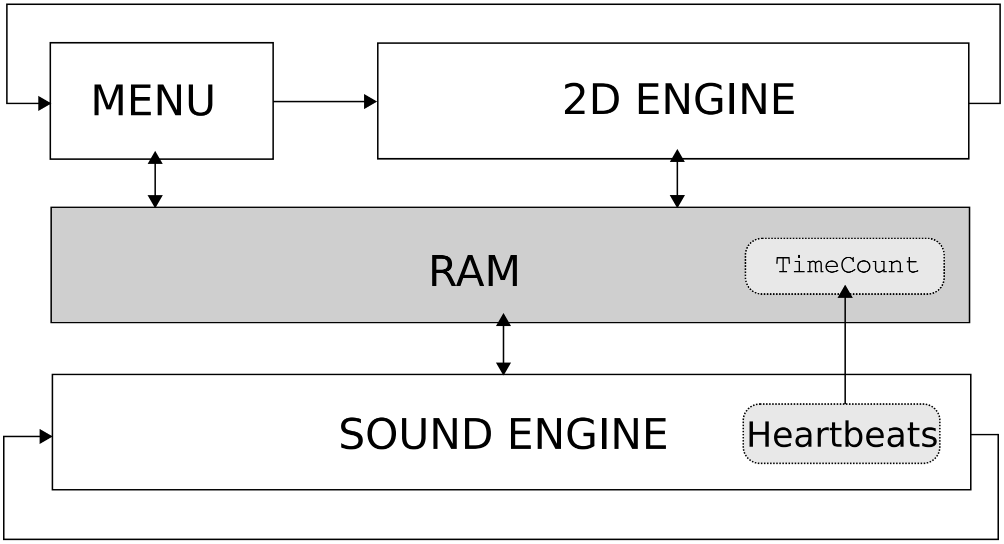

Figure 4: Game engine three main systems.

0.5.1 Unrolled Loop

With the big picture in mind, we can dive into the code and unroll the main loop starting in `void main()`. The two renderers are regular loops but due to limitations explained later, the sound system is interrupt-driven and therefore out of `main`. Because of real mode, C types don't mean what people would expect from a 32-bit architecture.

- `int` and `word` are 16 bits.
- `long` and `dword` are 32 bits.

The first thing the program does is set the text color to light grey and background color to black.

```
void main (void)
{
  textcolor(7);
  textbackground(0);

  InitGame();

  DemoLoop();          // DemoLoop calls Quit when
   everything is done
  Quit("Demo loop exited???");
}
```

In `InitGame`, a valdation is performed to check if suficient memory is available and brings up all the managers.

```
void InitGame (void)
{
  int i;

  MM_Startup ();     // Memory Manager

  US_TextScreen();   // Show intro screen

  VW_Startup ();     // Video Manager
  RF_Startup ();     // Refresh Manager
  IN_Startup ();     // Input Manager
  SD_Startup ();     // Sound Manager
  US_Startup ();     // Font Manager

  CA_Startup ();     // Cache Manager
  US_Setup ();

  CA_ClearMarks ();   // Clears out all the marks

  CA_LoadAllSounds (); // Load all sounds

}
```

Then comes the core loop, where the menu and 2D renderer are called forever.

```
void DemoLoop() {
    US_SetLoadSaveHooks();
    while (1) {
        VW_InitDoubleBuffer ();
        IN_ClearKeysDown ();
        VW_FixRefreshBuffer ();
        US_ControlPanel ();  // Menu
        GameLoop ();
        SetupGameLevel ();
        PlayLoop () ; // 2D renderer (action)
    }
    Quit("Demo loop exited???");
}
```

PlayLoop contains the 2D renderer. It is pretty standard with getting inputs, update world, and render world approach.

```
void PlayLoop (void)
{
  FixScoreBox ();     // draw bomb/flower
  do
  {
    CalcSingleGravity ();   // Calculate gravity
    IN_ReadControl(0,&c);   // get player input

    // go through state changes and propose movements
    obj = player;
    do
    {
      if (obj->active)
        StateMachine(obj);  // Enemies think

      obj = (objtype *)obj->next;
    } while (obj);

    [...]               // Check for and handle collisions
                        // between objects

    ScrollScreen();     // Scroll if Keen is nearing an edge
    RF_Refresh();       // Update the screen
  } while (!loadedgame && !playstate);
}
```

The interrupt system is started via the Sound Manager in SDL_SetIntsPerSec(rate).

While there is a famous game development library called Simple DirectMedia Layer (SDL), the prefix SDL_ has nothing to do with it. It stands for SounD Low level (Simple DirectMedia Layer did not even exist in 1991).

The reason for interrupts is extensively explained in Chapter **??** "**??**". In short, with an OS supporting neither processes nor threads, it was the only way to have something execute concurrently with the rest of the engine.

An ISR (Interrupt Service Routine) is installed in the Interrupt Vector Table to respond to interrupts triggered by the engine.

```
void SD_Startup(void)
{
  if (SD_Started)
    return;

  t0OldService = getvect(8);  // Get old timer 0 ISR

  SDL_InitDelay();  // SDL_InitDelay() uses t0OldService

  setvect(8,SDL_t0Service);   // Set to my timer 0 ISR

  SD_Started = true;

}
```

0.6 Architecture

The source code is structured in two layers. KD_* files are high-level layers relying on low-level ID_* sub-systems called Managers interacting with the hardware.

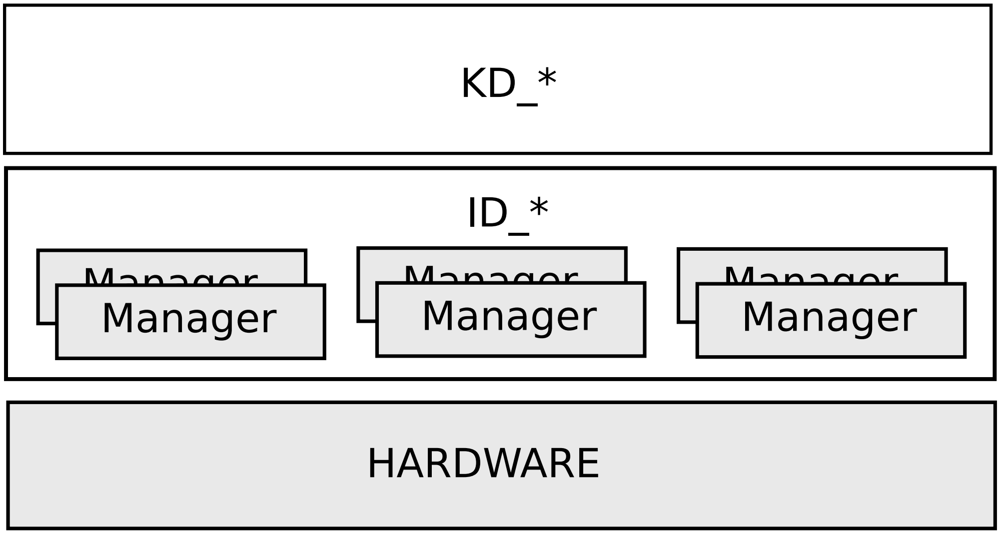

Figure 5: Wolfenstein 3D source code layers.

There are six managers in total:

- Memory
- Video
- Cache
- Sound
- User
- Input

The KD_ stuff was written specifically for Commander Keen while the ID_ managers are generic and later re-used (with improvements) for newer ID games (Hovertank One, Catacomb 3-D and Wolf3D).

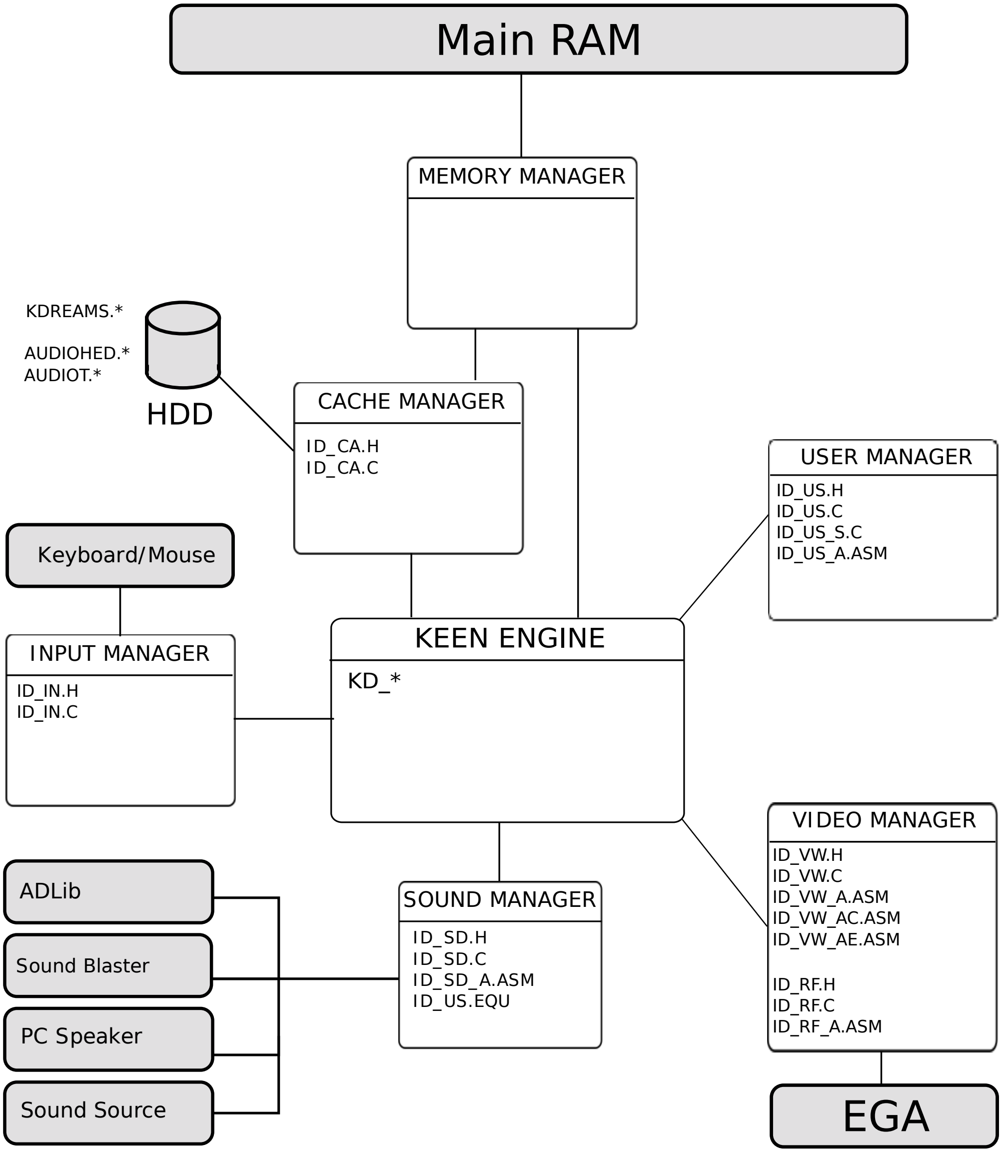

Figure 6: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

Next to the hard drives (HDD) you can see the assets packed as described in Chapter **??**.

0.6.1 Memory Manager (MM)

The engine does not rely on `malloc` to manage conventional memory, as this can lead to fragmented memory and no way to compact free space. It has its own memory manager made of a linked list of "blocks" keeping track of the RAM. A block points to a starting point in RAM and has a size.

```
typedef struct mmblockstruct
{
  unsigned   start,length;
  unsigned   attributes;
  memptr     *useptr;
  struct mmblockstruct far *next;
} mmblocktype;
```

A block can be marked with attributes:

- `LOCKBIT` : This block of RAM cannot be moved during compaction.
- `PURGEBITS` : Four levels available, 0= unpurgeable, 1= purgeable, 2= not used, 3= purge first.

The memory manager starts by allocating all available RAM via `malloc/farmalloc` and creates a `LOCKED` block of size 1KiB at the end. The linked list uses two pointers: `HEAD` and `ROVER` which point to the second to last block.

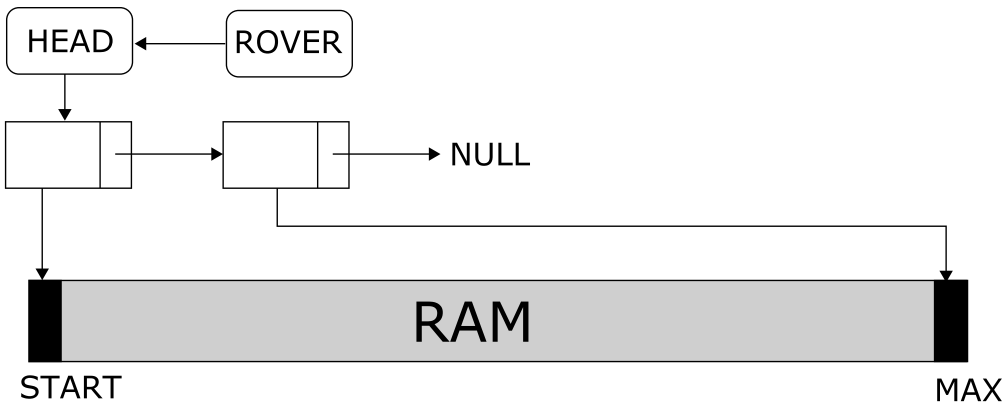

Figure 7: Initial memory manager state.

The engine interacts with the Memory Manager by requesting RAM (`MM_GetPtr`) and freeing RAM (`MM_FreePtr`). To allocate memory, the manager searches for "holes" between blocks. This can take up to three passes of increasing complexity:

1. After rover.
2. After head.
3. Compacting and then after rover.

The easiest case is when there is enough space after the rover. A new node is simply added to the linked list and the rover moves forward. In the next drawing, three allocation requests have succeeded: `A`, `B` and `C`.

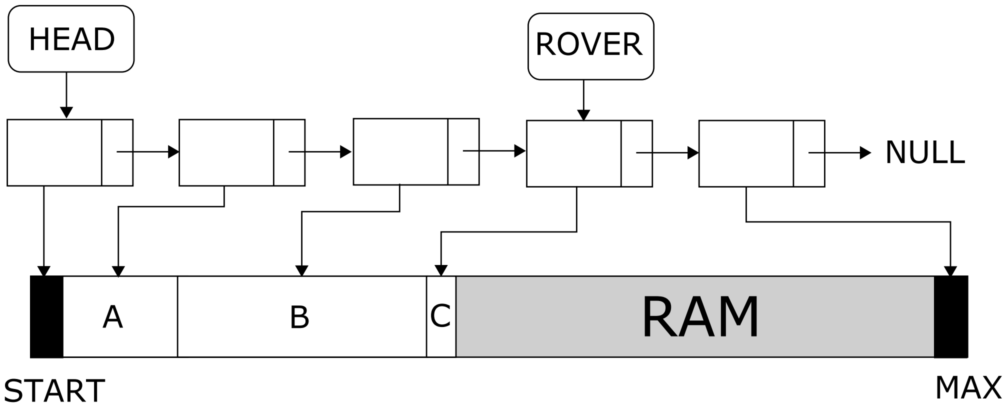

Figure 8: MM internal state after three pass 1 allocations.

Eventually the free RAM will be exhausted and the first pass will fail.

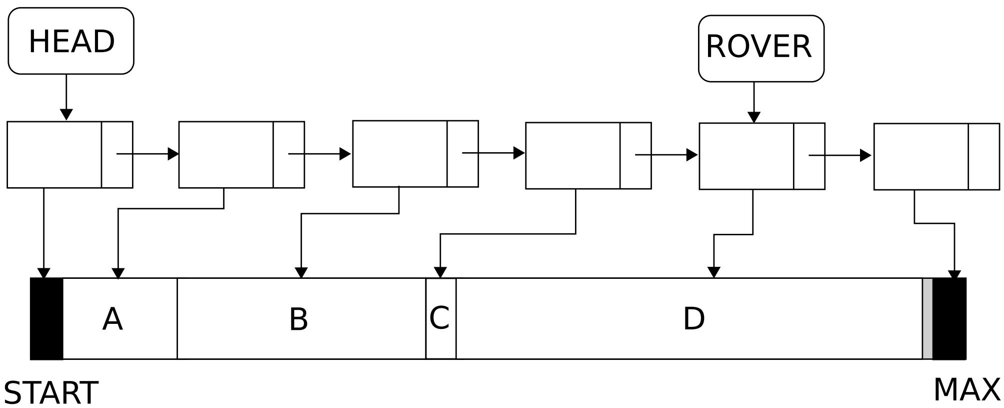

Figure 9: Pass 1 failure: Not enough RAM after the ROVER.

If the first pass fails, the second pass looks for a "hole" between the head and the rover. This pass will also purge unused blocks. If for example block B was marked as `PURGEABLE`, it will be deleted and replaced with the new block E. At this point fragmentation starts to appear (like if `malloc` was used).

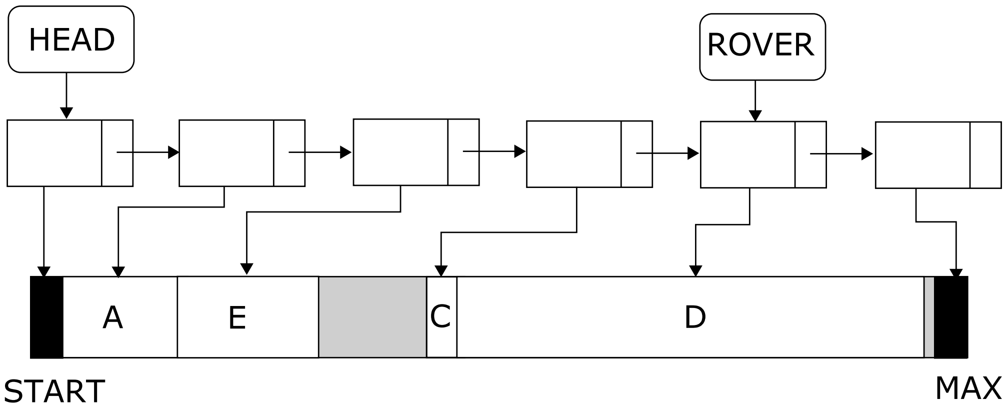

Figure 10: `B` was purged. `E` was allocated in pass 2.

If the first and second pass fail, there is no continuous block of memory large enough to satisfy the request. The manager will then iterate through the entire linked list and do two things: delete blocks marked as purgeable, and compact the RAM by moving blocks.

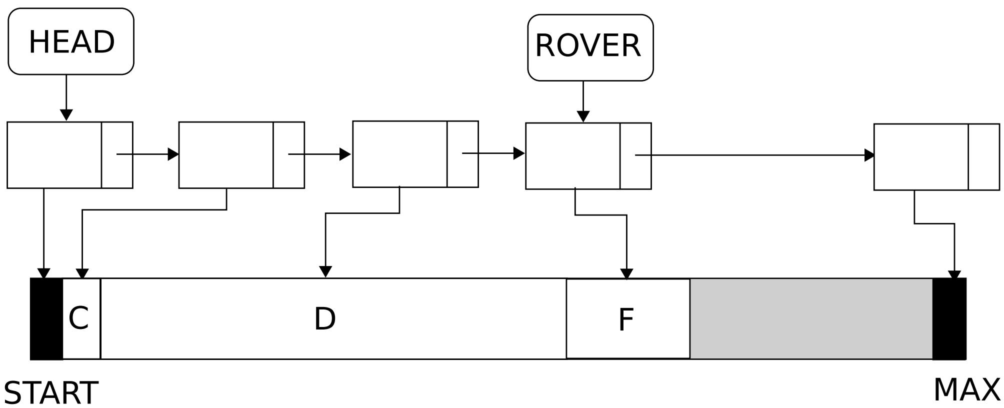

Figure 11: `A` and `E` were purged. `C` and `D` compacted. `F` allocated in pass3.

But if memory is moved around, how do previous allocations still point to what they did before the compaction phase? Notice that a `mmblockstruct` has a `useptr` pointer which points to the owner of a block. When memory is moved, the owner of the block is also updated.

As some blocks are marked as `LOCKED`, compacting can be disturbed. Upon encountering a locked block, compacting stops and the next block will be moved immediately after the locked block, even if there was space available between the last block and the locked block.

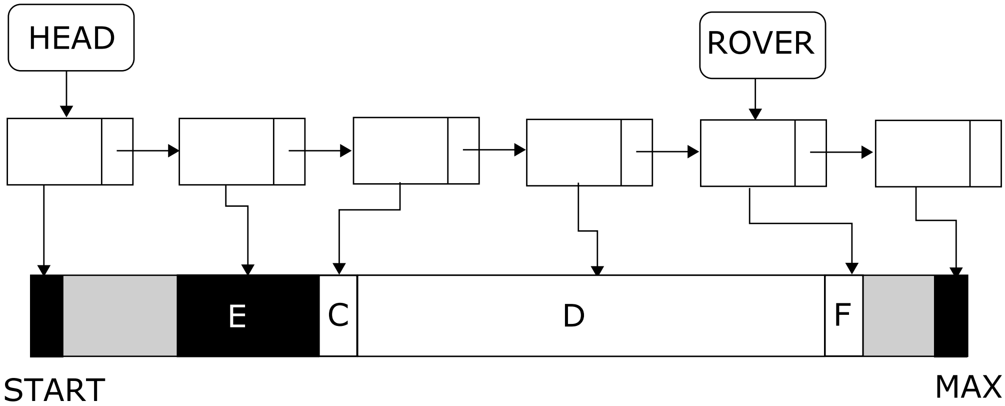

Figure 12: `E` is locked and cannot be compacted.

In the above drawing, C was moved after E, even though it could have been moved before. Avoiding this waste would have made the memory manager more complicated, so the waste was deemed acceptable. Often in designing a component you have to be practical and establish a certain trade off between accuracy and complexity.

0.6.2 Video Manager (VW & RF)

The video manager features two parts:

- The `VW_*` layer is made of both C and ASM, where the C functions abstract away EGA register manipulation via assembly routines.
- The `RF_*` layer is used to update tiles, and is also made both C and ASM code.

0.6.3 Sound Manager (SD)

The Sound Manager abstracts interaction with all four sound systems supported: PC Speaker, AdLib, Sound Blaster, and Disney Sound Source. It is a beast of its own since it doesn't run inside the engine. Instead it is called via IRQ at a much higher frequency than the engine (the engine runs at a maximum 70Hz, while the sound manager ranges from 140Hz to 700Hz). It must run quickly and is therefore written with small and fast routines.

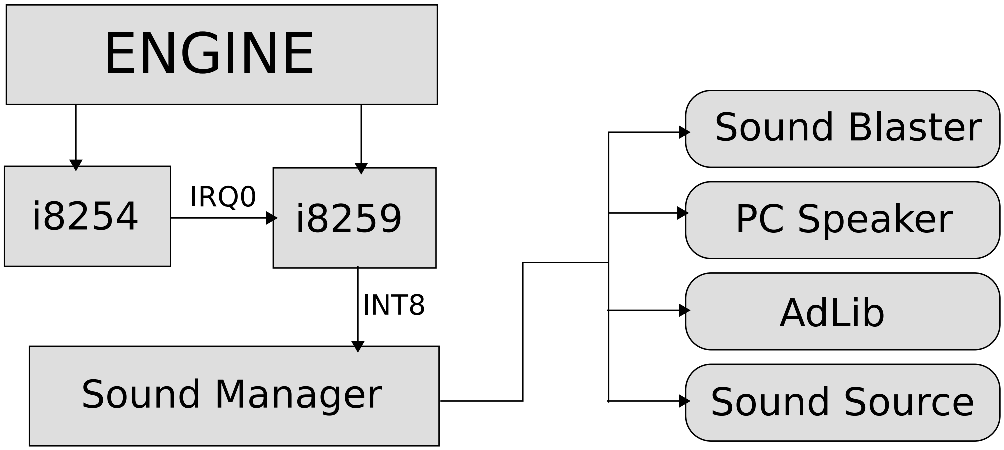

Figure 13: Sound system architecture.

The sound manager is described extensively in the "Sound and Music" section.