A Java Sega Master System and Game Gear Emulator

2001 - 2002

School of Computer Science
University of Birmingham

Written by
Christopher White
BSc Artificial Intelligence & Computer Science

Supervised by
Antoni Diller

1 PREAMBLE

1.1 - Abstract

The objective of this project was to produce the first Java based emulator capable of executing titles written for two separate games systems, (the Sega Master System and Game Gear). High performance, compatibility and accuracy were deemed key requirements in terms of the final product. Attention was paid to the central processing unit (CPU), video display processor (VDP) and programmable sound generator (PSG). Existing emulators are typically coded in low-level languages including assembler and C, which do not offer platform independence. Coding an emulator in Java represents a significant challenge because it is an interpreted language, unable to meet the speed of natively compiled code.

The emulator permits the operation of Master System and Game Gear titles on the Java platform. The product is considered to be highly reliable in terms of the key requirements highlighted above, when deployed as either an application or applet. From the 200 titles tested with the emulator, compatibility was achieved with 92% of the sample. Although the implementation was a complex programming task, a user friendly approach has been adopted in terms of the interface and emulator operability.

The following report examines the techniques available to emulate the hardware components of a complete computer system. The focus is directed towards the emulation of specific hardware. However, the principles applied and algorithms developed could form the basis for the emulation of a wider variety of hardware.

1.2 - Acknowledgements

I would like to thank the following people who have made this project possible:

- Antoni Diller for kindly supervising the project.
- The S8-Dev Forum for providing detailed technical information and answering my questions.
- 'Maxim' for his extensive e-mails explaining the inner workings of the Master System sound processor.
- Sean Young for his excellent 'Undocumented Z80 Documented' research.
- Charles MacDonald for his rigorous analysis of the Master System video processor.
- Sega, for continually producing the finest consoles and video games.

1.3 – Table of Contents

1.4 – Index of Figures

1.5 – Index of Tables

2.1 - A Brief History of Emulation

Emulation allows software written for one computer to be executed on a different computer, with identical results. An emulator is the name of the piece of software and/or hardware that is required to achieve this.

Emulation is not a new concept; the first emulator, the 7070 Emulator, was developed by Larry Moss of IBM in 1964. Comprising of hardware and software, it enabled older IBM 7070 mainframe programs to run on high-end System/360 computers. It was not until 1980 that an emulator for the personal computer was developed. This was Microsoft's Z80 SoftCard for the Apple II, which enabled it to run CP/M, the dominant business operating system of the day. This emulator consisted of a hardware card containing a real Z80 processor, as the Apple II was not powerful enough to support a software based emulator (Figure 2.1) [1]. In 1986, Avant-Garde Systems released PCDitto for the Atari ST, which emulated the IBM PC. Unlike previous emulators that required additional hardware, it was entirely software based, using native system resources, which made the emulation very slow.

Figure 2.1 – The SoftCard

In recent years, as processor speeds have increased, software emulators have become more viable. This, combined with the growth of the Internet has triggered the development of hundreds of emulators, both commercial and amateur, for almost every system in existence, from calculators to arcade machines.

2.2 - The Uses of Emulation

Emulation is actively used when prototyping and developing new hardware. In the late 1970s, Intel began using emulation to verify chip designs as chips became increasing complex, containing over a million transistors. Intel had previously used "breadboards" for testing; a board containing commercial logic components to simulate the final design. Initially emulation was run in parallel with the breadboard simulations. The older breadboard techniques were found to use five times as many employees, as well as more money, equipment and lab space [2]. The commercial benefits of emulation in prototyping are obvious, as a new system can be designed and debugged without the cost of building a single chip.

Emulation has also been widely utilised in software development. A recent example is the mobile phone emulator, which is packaged with Nokia developer's suite for Java [3]. When compiling software for a remote device, it is easier to test the software locally, as opposed to uploading a binary to a separate device. In addition, debugging software is a simpler process under emulation. Emulators used for development purposes typically allow software to be executed at a very low-level, instruction by instruction. The exact register and flag changes produced in the CPU can be examined, allowing bugs to be tracked and fixed. Memory can be dumped and reloaded at any stage of execution, effectively preserving the state of the system at any stage for cross-examination and later restoration. Graphics can be dumped and even adjusted during runtime, resulting in

immediate change, whilst avoiding recompiling. Emulators simplify the software development process, offering functionality that would often be impossible on the original hardware.

Cross platform development is simplified by emulation. A prominent example is the Java Virtual Machine (JVM). The JVM emulates a machine that only exists in software. The source code is compiled to an intermediate format, known as bytecode, which is translated to the target format by the virtual machine at runtime. This enables Java applications to run on multiple platforms, and also solves problems that have traditionally hindered cross platform development, including maintaining a consistent user interface between platforms.

Emulation serves as a virtual archive of obsolete hardware. For example, the popular emulator MAME, (the Multiple Arcade Machine Emulator) emulates over 1000 arcade machines, dating from 1975 to the present day [4]. The majority of these games no longer exist in arcades, having been superseded shortly after their release. Even if they did still physically exist, the hardware is unlikely to be functioning after 25 years of use. It might be argued that MAME serves no practical purpose, apart from satisfying nostalgic gamers eager to recapture their youth, but it remains the most successful recreational example of emulation in use today.

Emulation is used to maintain backwards compatibility, an aspect that is often exploited commercially by hardware manufacturers when encouraging customers to upgrade to the latest iteration of their product. Sony's Playstation 2 console uses a combination of hardware and software to run existing Playstation 1 titles. Sega licensed an existing free MegaDrive emulator called KGen, to bring MegaDrive titles to their current Dreamcast console (Figure 2.2). Previously, a hardware emulator called the PowerBase enabled the real MegaDrive to run Sega Master System games. Consequently, emulation in video gaming offers more than novelty value; it provides a smooth transition between platforms and maintains customer loyalty.

Figure 2.2 –Sega Smash Pack

2.3 - Legal Aspects of Emulation

Emulating a system will involve reverse engineering methods. The Copyright (Computer Programs) Regulations 1992 state that it is lawful to decompile a program "to obtain the information necessary to create an independent program which can be operated with the program decompiled or with another program" [5]. The regulations also permit "the use of any device or means to observe, study or test the functioning of the program in order to understand the ideas and principles which underlie any element of the program." [5]

For effective use, emulators require a copy of software written for the original machine. The Copyright Regulations state "It is not an infringement of copyright for a lawful user of a copy of a computer program to make any back up copy of it which it is necessary for him to have for the purposes of his lawful use." [5] Therefore, the distribution of an emulator is legal, but the distribution of proprietary software, including that contained within the BIOS of the system, is considered unlawful. Users must back up their own software for use with an emulator.

2.4 - Sega Master System and Game Gear History

The Sega Master System is a dedicated game-playing device, otherwise known as a console. It connects to a television and allows up to two people to play a variety of games, available separately in cartridge format. The Master System is an easy device to operate, especially when compared to a computer (Figure 2.3) [6]. Playing a game simply involves inserting a cartridge and turning the system on.

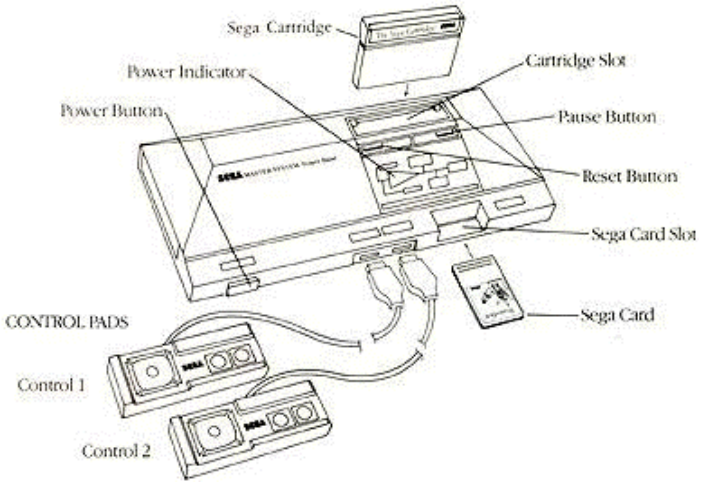


Figure 2.3 – Sega Master System Overview

The Master System was first launched by Sega of America in June 1986, in an attempt to break Nintendo's dominance over the home video game market (Figure 2.4). It offered superior specifications to its closest rival, the Nintendo Entertainment System (NES), boasting more main memory and 8 times as much video memory, as well as the more powerful Zilog Z80 CPU. Despite a promising launch, the well established NES continued to outsell the Master System by a sixteen to one ratio. The European launch in 1987 proved more successful; the Master System represented the first console to be widely marketed in Europe and it rapidly became the dominant machine. Whilst initial titles were limited to those previously developed in Japan and America, European software developers soon produced hundreds of new titles for the machine, many of which proved to be more popular than their foreign counterparts.

Figure 2.4 – Sega Master System I

Figure 2.5 - Sega Master System II

In 1990, Sega redesigned the Master System, streamlining its appearance and cutting costs by removing unnecessary components, producing the Master System II (Figure 2.5). Sega had now firmly established themselves across Europe as a major presence in home entertainment due to the success of the Master System. The system's final incarnation, the Master System III, was launched in Brazil shortly afterwards, under license to Tec Toy, and remained on sale until 1997. This model was essentially the same as the Master System II, but encompassed localisation features for the Brazilian market, and in-built games. In total, over 13 million Master System consoles were sold between 1986 and 1998, and over 400 games produced for the system.

Whilst the Master System II was being launched, the same technology was being recycled by Sega for use in its Game Gear console, Sega's answer to Nintendo's Game Boy (Figure 2.6). It is a portable, hand held machine designed to be technically superior to its rival in every possible way. Whilst physically, the Master System and Game Gear could not be more different, the only real technical

Figure 2.6 – Sega Game Gear

difference is the Game Gear's improved graphical capabilities, offering a palette of 4,096 colours, compared to the Master System's palette of 64. Nevertheless, while some titles were directly ported between the two systems, the Game Gear saw many new and innovative titles. Additionally, hardware converters were released that allowed Game Gear owners to make use of existing Master System cartridges. The Game Gear did not match the success of the Game Boy, mainly due to the Game Boy's smaller size and superior battery life, but it did enjoy moderate success. Sega stopped manufacturing the system in 1996, and it is now manufactured by Majesco, marketed as a low cost competitor to the Game Boy.

2.5 - Existing Emulators

There are currently around 15 Sega Master System and Game Gear emulators for a variety of platforms. The state of these emulators is varied; the best offer close to full compatibility with the original hardware, whilst others are still at an early stage of development. The most complete emulator, Meka [7], boasts high compatibility and a well designed, skinnable interface (Figure 2.7). It also supports a homemade hardware adapter, enabling original Sega peripherals like 3D glasses to be used. However, Meka is only available for DOS, and requires a payment if used regularly.

Figure 2.7 – Meka Emulator for DOS

Unlike Meka, SMS Plus [8], is free and open source. It has been ported to a variety of platforms, including DOS, Windows and the Dreamcast console. The user interface code is left to the authors maintaining the individual ports, in order to retain code portability. The interface is therefore not as sophisticated as Meka's, nor does it provide any debugging information.

Coding in Java overcomes the cross platform limitations present in these existing emulators. JavaGear is compatible with any platform capable of running the JVM, as opposed to being tied to a single platform. Swing, the Java GUI toolkit, maintains a consistent interface between platforms avoiding the issues present when porting C interface code. Java has proved a popular emulation platform in recent years due to its platform independence, despite its speed deficiencies in contrast with native code.

NESCafe, the most refined Java based emulator to date, emulates the NES console with impressive speed and accuracy [9]. It was the first emulator to implement sound emulation in Java, supports additional peripherals and has a coherent interface. A further example to note is GameBoyEmu, a Game Boy emulator programmed by a former Birmingham student [10]. There is no sound emulation and emulation accuracy is comparatively low when compared to other Java GameBoy emulators, like JavaBoy [11]. However, GameBoyEmu offers an

excellent user interface and sophisticated debugging facilities, rendering it a worthy competitor.

2.6 - JavaGear

JavaGear emulates both the Sega Master System and Sega Game Gear consoles. At the simplest level, JavaGear takes a binary for one of the emulated systems and executes it with identical results to the original hardware. In order to achieve this, JavaGear emulates the key components of the two systems, the Zilog Z80 Central Processing Unit (CPU), memory paging hardware, custom video display processor (VDP) and Texas Instruments programmable sound generator (PSG). In addition, JavaGear provides code debugging facilities not present in the original hardware.

JavaGear masks its internal complexity with a simple GUI, reflecting the ease of use present in the original systems, often designed with children in mind. Furthermore, JavaGear can either be run as an application, or embedded as an applet, in a webpage. This eliminates the need to download and configure the emulator, resulting in browser based operability.

This is a particularly interesting project as its scope encompasses areas from CPU design, to display generation and sound synthesis. The decision to code the emulator in Java renders the project even more challenging, as the JVM is an emulator in itself, so an emulator is effectively being run on an emulator. This places the onus on code efficiency, as Java is an interpreted language and typically slower than natively compiled code. JavaGear's algorithms must not only be accurate, but should also be highly optimised to compensate for this.

The challenge in coding a project of this nature is in endeavouring to ensure the emulation is as accurate as possible. A proprietary gaming system like the Master System is not officially documented anywhere. Its specifications and behaviour must be ascertained from a variety of sources on the Internet. This includes the analysis of binaries and source code for the system, and reference to amateur programming documentation and development forums. There are no commercial books covering emulation in general, which makes this an intriguing area of research. This report will analyse the hardware present in the Master System and Game Gear and discuss its optimal software implementation.

3.1 – Hardware Overview

Figure 3.1 [12] provides an overview of the hardware components present in the Master System.

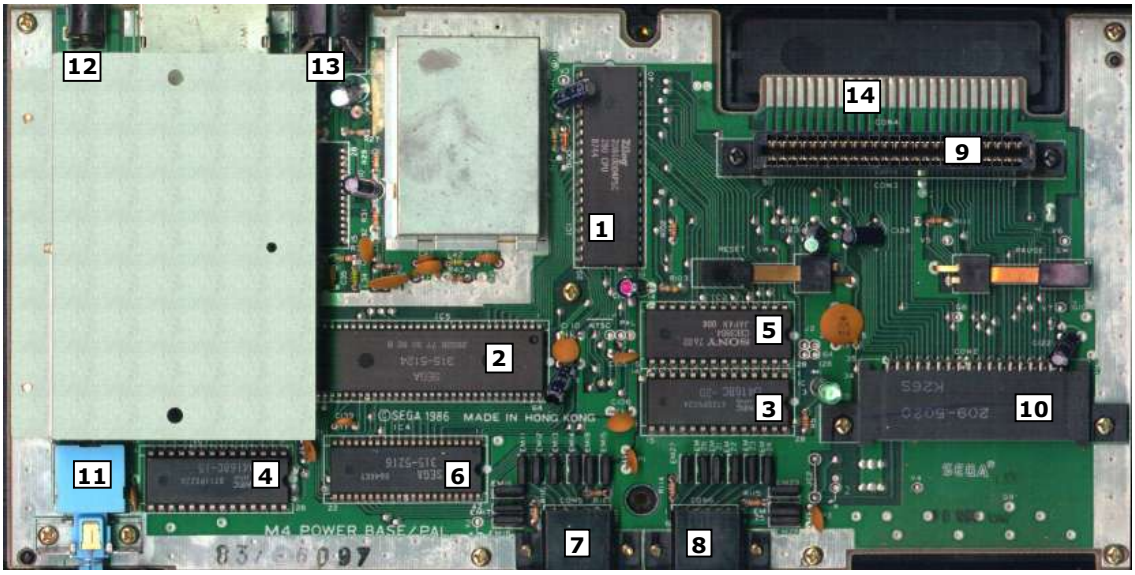


Figure 3.1 – Sega Master System Board

Key

1	Zilog Z80 CPU (3.58MHz)	**7**	Control Port 1
2	Custom Video Processor (VDP) & Texas SN76489 Programmable Sound Generator (PSG)	**8**	Control Port 2
		9	Cartridge Slot
		10	Card Slot
3	8K RAM	**11**	Power Button
4	16K VRAM	**12**	AC Adapter Socket
5	Sega BIOS ROM	**13**	Video Cable Socket
6	Controller I/O	**14**	Extension Port

The original Master System board is similar in design, if not appearance to the Master System II and the Game Gear. The components are virtually identical, making an in-depth analysis of the hardware inappropriate. The key difference between the models is the version of the VDP deployed by Sega. Initial bugs are corrected in later iterations of the processor. Physically, the Master System II board is more compact due to the omission of certain components to cut manufacturing costs. These include the card slot and extension port, which were rarely used on the original model. The Game Gear differs in that the CPU, VDP and sound generator are combined into a single ASIC chip to reduce the size of the unit.

From this point onwards, the term 'Master System' will refer to both the Master System and Game Gear, unless explicitly stated. The term 'Game Gear' will be used to denote aspects of functionality not present in the Master System.

3.2 – Z80 CPU

The Zilog Z80 CPU was first released in July 1976. It was an immediate success due to its backwards compatibility with the Intel 8080, running existing software without modification. The Z80 proved a popular choice for computers, consoles and other devices and was widely deployed in systems including the Sinclair Spectrum, Pacman arcade machine and TI-8x calculator range. By 1990, as gaming systems demanded increasingly powerful processors, the Z80 was commonly utilised as a secondary processor controlling sound hardware. The Z80 is still in production today; the architecture remains identical to the original model, but the clock speed has increased from 2.5MHz to 20MHz.

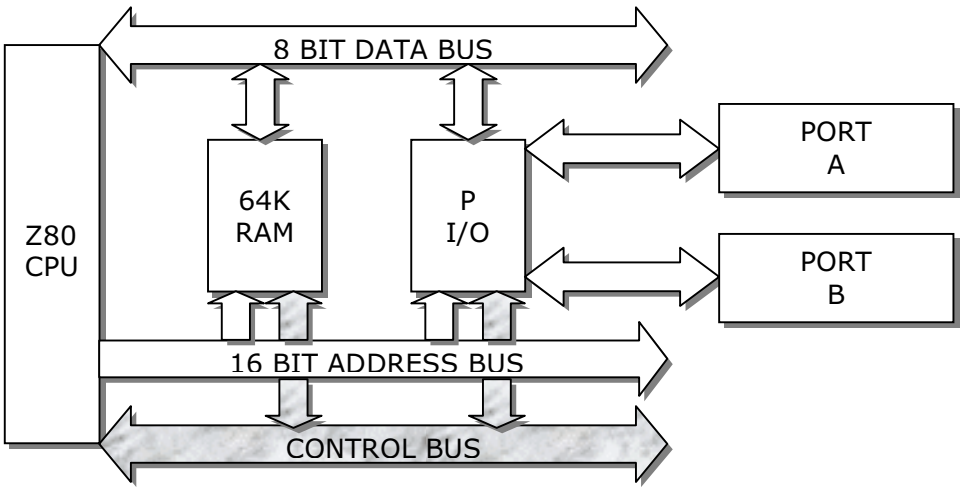


Figure 3.2 – Standard Z80 System

The Z80 is an 8-bit processor, comprising an 8-bit bidirectional data bus, a 16-bit unidirectional address bus and a control bus (Figure 3.2) [13]. The data bus transports data between the Z80 and the system's memory or an external device. The address bus specifies a source or destination address for this data, a 16-bit number, providing access to 64K of memory. The control bus facilitates the internal synchronisation of the system.

3.2.1 – Registers

The Z80 has a series of registers, which it uses to store data when performing its main operations. They provide a more efficient method to manipulate data than by addressing slower, external memory. Registers are comparable to variables in a programming language. The Z80 has a combination of 8-bit and 16-bit registers, which follow:

General Purpose Registers

- **A** *The Accumulator*
  The Accumulator is used for 8-bit arithmetic and Boolean operations. The shortest and fastest data transfers between the CPU and I/O devices are performed through the Accumulator.

- **HL** *Primary Data Pointer*
  The HL pair is normally used to store the 16-bit memory address of data being accessed. It is directly connected to the address bus, making it ideal for operations reliant on manipulating memory.

- **BC**, **DE** *Secondary Data Storage*
  There are a small number of instructions that also use these registers as 16-bit pointers, but they are primarily used for the temporary storage of data and addresses.

The general purpose registers can be accessed and manipulated individually, or combined, as a register pair. Individually, the registers hold an 8-bit value and when combined they hold a 16-bit value. Table 3.1 illustrates this property; the H register contains the high byte and the L register contains the low byte of the pair. This property is clearer when using hexadecimal notation.

	H	L	HL
Decimal Value	25	70	6470
Hex Value	0x19	0x46	0x1946

Table 3.1 – Register Pairs

- **A', BC', DE', HL'** *Alternate Registers*
  The alternate registers provide a duplicate bank of general purpose registers that can be exchanged with the originals at anytime. One bank acts as an internal memory, while the other behaves as a working set of internal registers.

Address Registers

- **IX, IY** *Index Registers*
  The two 16-bit index registers can be used interchangeably, and as a substitute for the HL register pair. The index registers, however, support a displacement value that is added to the index address. This asserts their use to access elements of a table successively in an efficient way. Officially, the index registers cannot be manipulated as two 8-bit values like the general purpose registers, but undocumented instructions exist to use them in this manner. The index registers are used less frequently than the general purpose registers, as accessing them is slower.

- **PC** *Program Counter*
  The program counter contains the 16-bit address of the next instruction to be executed.

- **SP** *Stack Pointer*
  The stack pointer contains the 16-bit address of the top of the stack within memory. The stack grows 'downwards' in memory towards the lower addresses.

Other Registers

- **I** *Interrupt Page Address Register*
  This 8-bit register was designed to store the partial address of an interrupt routine in memory. The low byte of the address is provided by the device generating the interrupt and the I register provides the high byte. The Master System does not use the interrupt register in this manner; instead, it is occasionally used to store a copy of the Accumulator.

- **R** *Memory Refresh Register*
  This 8-bit register is automatically incremented after each instruction executed by the processor. It is intended to facilitate the use of dynamic memory with the processor. The Master System does not use the refresh register in this manner; instead, it is occasionally used as a random number generator.

- **F** *Flag Register*

The flag register stores certain conditions after an instruction is executed. Program flow relies on this register; all jumps executed within a program are dependent on the status of these flags. It is an 8-bit register, each bit represents a particular condition:

Bit	7	6	5	4	3	2	1	0
Flag	Sign	Zero	Bit 5	Half-Carry	Bit 3	Parity	Negative	Carry

Table 3.2 – Flag Register

The flag register (Table 3.2) must be emulated in an identical manner to the original Z80. Failure to set this register correctly after each instruction will disrupt program flow; the slightest error will mean the software does not function.

Sign Flag

The sign flag reflects the most significant bit of an arithmetic or logic operation. In two's complement notation, the most significant bit is used to represent the sign. '0' indicates a positive number and '1' indicates a negative number.

In the example (Table 3.3), 10011011 could represent 155 in unsigned notation or –101 in two's complement notation. The Z80 cannot establish which representation is being used. The sign flag could be set when the number is actually positive, it is up to the programmer to interpret the flag correctly.

Result (Binary)	Sign Flag
00011011	**0**
10011011	**1**

Table 3.3 – Sign Flag Example

Zero Flag

The zero flag is set when the result of an operation, or a particular match, is zero, otherwise it is reset.

Bit 5 Flag

The bit 5 flag contains a copy of bit 5 of the result of an operation. It is unofficial and undocumented; its result is not interpreted by any instructions. Master System software studied was not dependent on this flag. It can be omitted from emulation as it serves no purpose and accurate emulation will incur a speed loss.

Half-Carry Flag

The half-carry flag is set when there is a carry from bit 3 to bit 4 during an addition operation or a carry from bit 4 to bit 3 during a subtraction operation (Table 3.4).

Hex	Binary
0x3A	0011 1010
0x7C	+0111 1100
0xB6	=1011←0110

Table 3.4 – Half-Carry Flag Example

Bit 3 Flag

The bit 3 flag contains a copy of bit 3 of the result of an operation. This flag is not emulated for the same reasons as the bit 5 flag.

Parity/Overflow Flag

The parity/overflow flag serves two functions. The first is storing the parity of a result; a method used to verify its integrity, particularly during I/O operations. The flag is set if the binary result contains an even number of 1s, and reset if the number is odd (Table 3.5).

Result (Binary)	Parity Flag
000**11**0**11**	**1**
100**11**0**11**	**0**

Table 3.5 – Parity Flag Example

It also serves as an overflow flag for values represented in two's complement notation. If the most significant bit, representing the sign, is accidentally changed by an arithmetic operation, the flag is set.

Decimal	Binary
127	0 1111111
1	+0 0000001
128	=**1**←0000000

Table 3.6 – Overflow Flag Example

In this example (Table 3.6), the sign has been changed and the flag is set. The final value would be –127, rather than 128 if two's complement notation was in use.

Subtract Flag

The subtract flag is set if the previous operation was a subtraction and reset if it was an addition. It is used internally by the Z80 when performing notation conversion operations.

Carry Flag

The carry flag is used to indicate whether an addition or subtraction operation has resulted in a carry or borrow. It effectively acts as a ninth bit and as a protection against overflow.

Hex	Binary
0xFF	11111111
0x01	+00000001
0x00	= (1) 00000000

Table 3.7 – Carry Flag Example

In this example, the carry flag is set as the result has exceeded the storage capacity of the 8-bit register. The carry flag is also used in shift and rotation operations to store a particular bit from a register.

3.2.2 – Opcodes

An opcode, or operation code, specifies the operation to be performed by the Z80. The strict definition of an opcode refers only to the operation, independent of the registers involved, but in microprocessor terminology it is convenient to incorporate the two. An opcode is encoded into a series of bytes, up to four consecutive bytes comprise a Z80 instruction. The Z80 has a total of 1,268 opcodes, both documented and undocumented, rendering accurate emulation a

significant challenge. Exact details of each opcode must be obtained, including the resultant flag changes from their execution.

The Z80, like any processor, operates in the following loop:

1 Fetch the next instruction
2 Decode the instruction
3 Execute the instruction

During the 'fetch' stage, the contents of the program counter are output on the address bus. The instruction at the specified address is then deposited on the data bus. The data bus is read and the instruction stored in an internal register. Once complete, the control unit decodes the instruction and generates the correct sequence of internal and external signals for its execution (Figure 3.3). The time the instruction takes to execute is dependent on whether it is executed internally, or requires external access to the system's memory.

PC	Memory Address	Opcode Fetched	Opcode Decoded	Execution Result
	0x13			
→	0x14	0x03	INC BC	Increment value stored in BC.
	0x15			
	0x16			

Figure 3.3 – Opcode Decoding

An emulated CPU will not naturally match the timings of the original CPU. However, it is desirable for the emulated CPU to execute the correct amount of instructions for the corresponding number of cycles it runs for. The Z80 present in the Master System operates at 3.58MHz, i.e. there are 3,580,000 cycles per second. The implementation must include an emulation of the machine cycles used by each operation. Exact opcode timing information is required to achieve this.

3.2.3 – Interrupts

The interrupt mechanism allows normal program flow to be suspended, and an interrupt-handling routine to be executed. On the Master System, an interrupt handling routine will typically contain code to read input from the game controllers, update elements of the screen display, and perform other routine operations.

Non-Maskable Interrupts

There are two interrupt mechanisms that will require emulation. The Non-Maskable Interrupt (NMI) has the highest priority of the two, so-called because it cannot be disabled by software. It is triggered by lowered the NMI pin of the Z80 chip. It is always recognised at the end of an instruction, and forces the following sequence to occur:

PC	→	STACK	Program Counter preserved on stack
IFF1	→	IFF2	Preserve pending maskable interrupts
0	→	IFF1	Further interrupts disabled
JUMP	→	0x66	Branch to address 0x66

The programmer restores the program counter and IFF1 at the end of the interrupt routine with appropriate instructions, so only the algorithm shown in the

table requires emulation. The Master System's pause switch is connected to the NMI pin of the Z80.

Maskable Interrupts

The second mechanism, the maskable interrupt, differs in that it can be disabled by the programmer. An interrupt is generated when an external device lowers the Z80's interrupt line and interrupts are enabled. According to the Master System schematics [14], the VDP is connected to the Z80's interrupt line, so maskable interrupts will be generated solely by this device. Exactly how and when it generates interrupts will be examined in section 3.3.5 (p.25).

The maskable interrupt can operate in one of three modes. Two of these modes are reliant on external hardware placing a value on the data bus to function correctly. The Master System does not utilise this feature and the value ends up defaulting to 0xFF. Therefore, only one of these three modes is actually of use, interrupt mode 1, which operates as follows:

PC	→	STACK	Program Counter preserved on stack
0	→	IFF1	Further interrupts disabled to prevent an infinite loop occurring.
0	→	IFF2	
JUMP	→	0x38	Branch to address 0x38

The programmer restores the program counter and re-enables interrupts at the end of the interrupt routine.

Interestingly, one Master System title, *Xenon 2*, operates in interrupt mode 0. Interrupt mode 0 executes an instruction from the data bus, in place of branching to 0x38. As mentioned, the default value on the data bus is 0xFF in the absence of a value from external hardware. This just happens to be the opcode RST 38h, which restarts the processor at address 0x38. It can be deduced that interrupt mode 0 operates identically to mode 1 on the Master System. A number of emulators cannot run *Xenon 2*, because this property is not emulated correctly.

3.3 – Custom Video Display Processor

The Master System VDP is a customised Texas Instruments TMS9918A processor. It is programmed through the Z80's I/O ports and is connected to 16K of dedicated video RAM (VRAM), separate from the Z80's address space (Figure 3.4). The VRAM is used to store graphical data and its on-screen location. The VDP has sixteen internal registers that are used to control display properties including display scrolling and sprite zooming. It also has 32 bytes of internal colour RAM (CRAM).

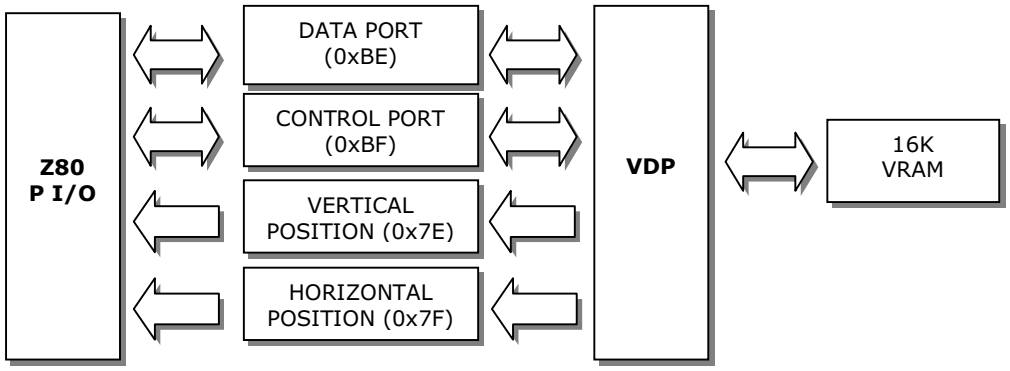


Figure 3.4 – VDP Access

3.3.1 - Ports

Control Port

The VDP is programmed through the control port. Two bytes must be written to the control port to define a particular operation (Table 3.8). The VDP has an internal flag to track which byte is currently being written.

	7	6	5	4	3	2	1	0
Byte 1	V7	V6	V5	V4	V4	V2	V1	V0
Byte 2	Op1	Op0	V13	V12	V11	V10	V9	V8

Table 3.8 – VDP Command Word

The operation performed is defined in the two most significant bits of the second byte (Op0 and Op1). The remaining bits specify an address or value for the operation. All operations configure subsequent data port access to address either VRAM or CRAM. Table 3.9 illustrates the results of the four operations.

Op0	Op1	Subsequent Data Port R/W	V0 – V13 Contents	Additional Operations Performed
0	0	VRAM	VRAM offset	Store byte of VRAM in read buffer Increment VRAM offset by 1
1	0	VRAM	VRAM offset	None
0	1	VRAM	VDP register value	Write to VDP register
1	1	CRAM	CRAM offset	None

Table 3.9 – VDP Command Word Operations

Data Port

VRAM and CRAM are accessed via the data port, after being configured through the control port. When a value is read from or written to the port, the offset of VRAM or CRAM is incremented automatically which allows it to be rapidly accessed. The value returned from the port is actually buffered, which prevents a delay in fetching the value from memory.

3.3.2 - Tiles

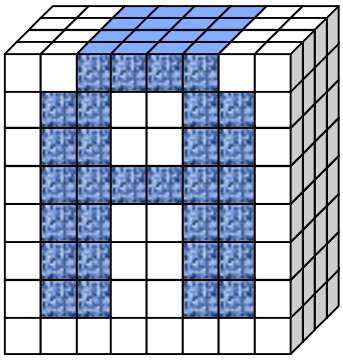


Figure 3.5 – An 8x8 Tile

All graphics on the Master System are made from 8x8 groups of pixels known as tiles. Each row of a tile contains four bit-planes, making the tile 32 bytes in length. The four bits that comprise a pixel contain its palette entry (0-15) from CRAM.

For example, the top row of the tile in Figure 3.5 would be defined as follows to use palette entry 15:

Byte 1	Byte 2	Byte 3	Byte 4
00**1111**00	00**1111**00	00**1111**00	00**1111**00

3.3.3 - CRAM

The VDP has 32 bytes of internal CRAM, which cannot be directly read. Instead, it is indirectly read by specifying a palette entry when defining tile pixels. As mentioned, the palette entry can only range between 0 and 15, so a special flag must be set to access entries between 16 and 31, which essentially acts as an extra bit. The format of each byte is as follows:

7	6	5	4	3	2	1	0
-	-	Blue	Blue	Green	Green	Red	Red

Table 3.10 – Master System Palette Byte

This means 64 individual colours can be generated, of which 32 can be displayed on screen simultaneously. Certain demos display 64 by interrupting program execution before the screen is fully displayed and writing to CRAM during this time period.

The Game Gear's VDP differs from the Master System's in that it contains double the amount of CRAM. Two bytes define each palette entry, instead of one. The format of each entry is as follows:

	7	6	5	4	3	2	1	0
Byte 1	Green	Green	Green	Green	Red	Red	Red	Red
Byte 2	-	-	-	-	Blue	Blue	Blue	Blue

Table 3.11 – Game Gear Palette Word

Each entry consists of 12 bits, allowing 4,096 individual colours to be generated. Palette entries are still indexed between 0 and 31; the hardware doubles the index value to compensate for the extra byte.

3.3.4 - Layers

The display is generated from two separate graphical layers, the background and sprite layers. These are combined as follows:

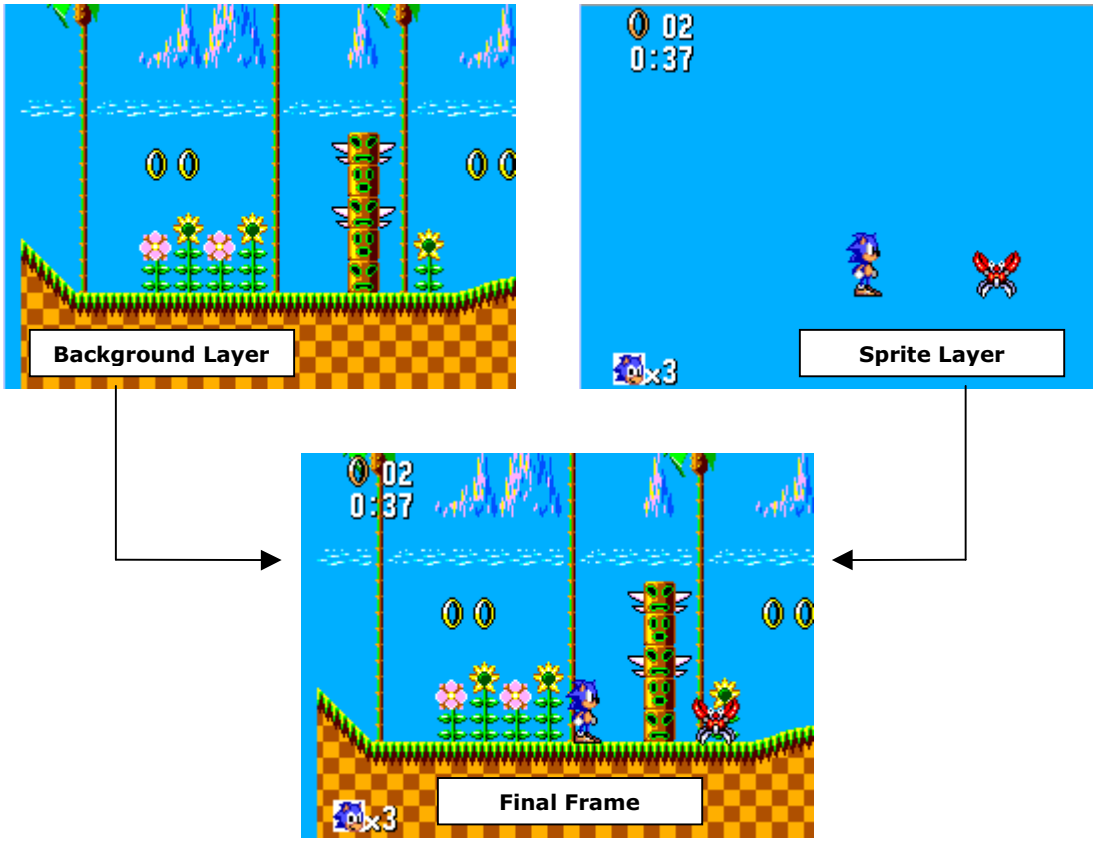


Figure 3.6 – Display Layers

Background Layer

The background layer consists of a 32x28 matrix of tiles. This implies graphical scenes have minimal memory requirements, as tiles can be repeated throughout the matrix. Using a matrix also reduces the load on the CPU, allowing tiles to be quickly swapped, without requiring the entire scene to be redrawn. The actual screen area is cropped to 32x24 as follows:

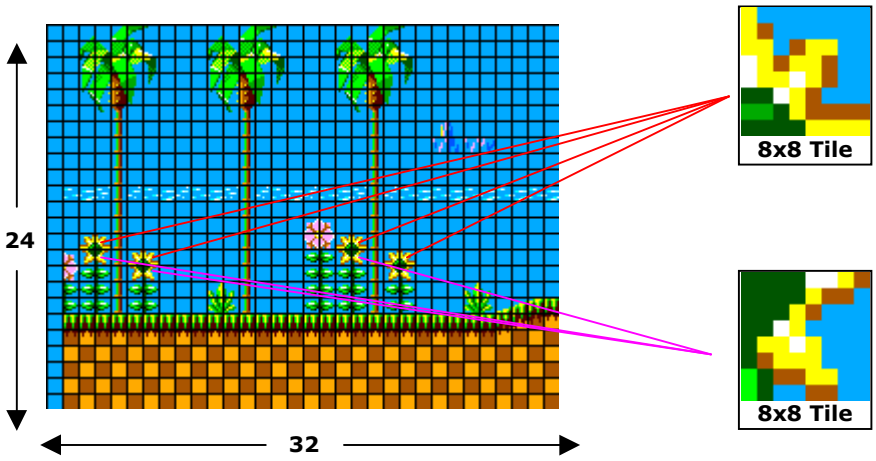


Figure 3.7 – 32x24 Background Layer Tile Matrix

Two bytes define the property of each tile in the matrix (Table 3.12).

	7	6	5	4	3	2	1	0
Byte 1	A7	A6	A5	A4	A3	A2	A1	A0
Byte 2	-	-	-	Priority	Palette	V Flip	H Flip	A8

Table 3.12 – Tile Properties

The bits A0 to A8 define the address of a tile in VRAM and the palette bit selects which of the two palettes to use. The tile can also be flipped vertically and horizontally by the appropriate bits, providing a method to maximise the use of VRAM, by avoiding tile repetition. The symmetrical nature of the flowers in Figure 3.6 is related to this particular use of horizontal tile flipping. The priority bit specifies the interaction between the background and sprite layer. When set, the tile in question will be displayed in front of the sprite layer, otherwise it will be displayed behind the sprite layer.

The background layer can be scrolled both horizontally and vertically in hardware. The background layer is offset by selecting a column at which to start rendering the layer. For example, an offset of 10 would generate column 0 in place of column 10, column 1 in place of column 11 etc. The layer wraps appropriately, so when the final column is reached, column 0 is displayed. A Master System test program was coded to demonstrate this property (Figure 3.9).

Column 0

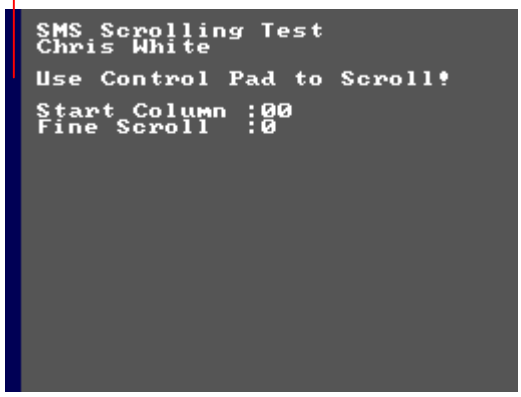


Figure 3.8 – No Horizontal Offset

Column 26 (0x1A)

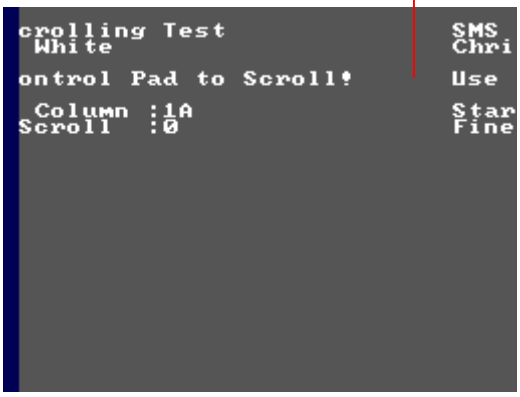


Figure 3.9 – Horizontal Offset of 26

A fine scroll value can also be specified, between 0 and 7, that smoothes the scrolling effect. Unlike most video hardware, the leftmost column does not contain graphics from the rightmost column as the fine scroll value increases. This leads to a jagged scrolling effect, so column 0 is typically blanked when performing horizontal scrolling, as in the example (Figure 3.8).

Vertical scrolling works in a similar manner, with the addition of an off screen area of four extra tiles. Tiles wrap in accordance with this off screen area (32x28), as opposed to the clipped display (32x24).

The VDP's internal registers are used to specify the horizontal and vertical scroll settings. Horizontal and vertical scrolling can be disabled for certain columns and rows. Many titles use this feature to scroll the display whilst simultaneously providing a static status bar. These properties allow the video hardware to generate complex scrolling effects, whilst freeing the CPU for other tasks.

Sprite Layer

Sprites are composed from tiles in the same manner as the background layer. They can be freely positioned on the screen, making them ideal for objects where greater precision is required, e.g. characters controlled by the user. Sprites are defined in a sprite attribute table. This table holds the properties of 64 sprites, their co-ordinates and the corresponding tile number to plot. The attributes can be extracted as follows:

Sprite Property	Location in Sprite Attribute Table
Sprite **X** Position	(Sprite Number * 2) + 0x80
Sprite **Y** Position	Sprite Number
Tile Number To Use	(Sprite Number * 2) + 0x81

The sprite hardware processes the sprite attribute table, drawing sprites that fall on the current scanline. Sprites with a lower number overprint those with a high number (Figure 3.10). The hardware will render a maximum of eight sprites per scanline; if there are remaining sprites to plot, a sprite overflow flag is set.

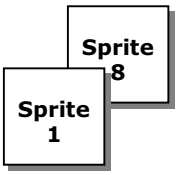


Figure 3.10 – Sprite Priority

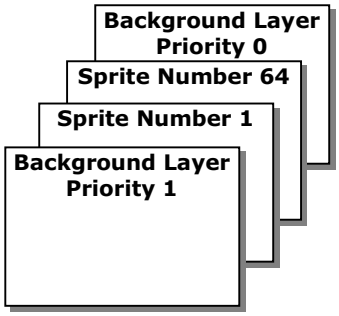


Figure 3.11 - Priorities

Figure 3.11 illustrates the priorities between the sprite and background layers. Layers with a low priority will only be displayed through layers of a high priority if the higher priority layer contains transparent pixels.

If two sprites have opaque pixels that overlap then a flag is set to indicate a sprite collision.

Sprites are not constrained to individual 8x8 tiles; 8x16, 16x16 and 16x32 sprites can also be generated by setting the appropriate registers.

3.3.5 – VDP Interrupts

The VDP can generate two types of interrupt, line interrupts and frame interrupts. Both types of interrupt can be enabled or disabled by the programmer. Frame interrupts are generated between rendering frames. Line interrupts are generated after rendering a specific number of lines, set by the programmer.

The VDP's interrupt settings are independent of the Z80's interrupt flip-flops. Therefore, the VDP's interrupts can be enabled whilst the Z80's are disabled. Figure 3.12 provides an example of this interaction. The VDP stores pending interrupts until the Z80 is ready to accept them.

Line interrupt frequency is set by loading a register with a value between 0 and 255. A counter is loaded with the register value, which decrements with every line rendered. When the counter expires, an interrupt is generated and the counter is reloaded.

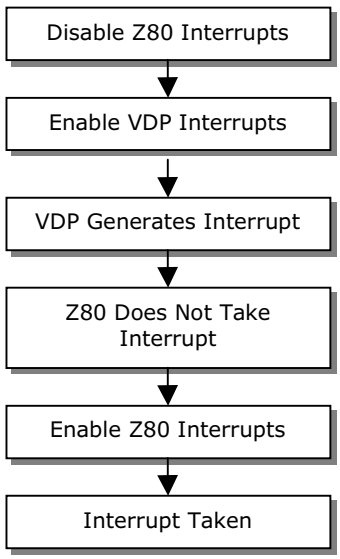


Figure 3.12 – Interrupt Independence

The interrupt line is cleared whenever the control port is read.

3.3.6 – Display Timing

The Master System hardware is capable of operating with both the NTSC and PAL television standards.

	Lines	Frames Per Second
NTSC	262	60
PAL	312	50

Table 3.13 – NTSC and PAL

A PAL display has 50 more lines than an NTSC display (Table 3.13). Despite this, the active display area, shown in the centre of Figure 3.13, remains fixed at 192 lines. Instead, a greater proportion of the screen is taken by the top and bottom borders, which has the effect of 'squashing' the active display.

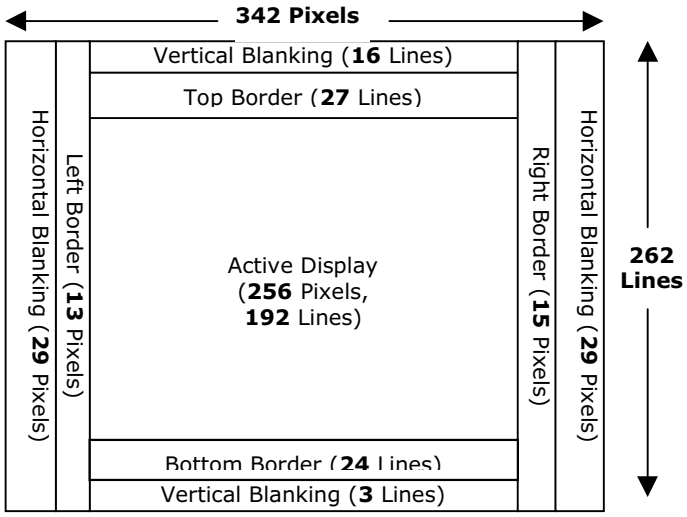


Figure 3.13 – NTSC Display

A television electron beam cannot move instaneously from one line to the next, a delay, (known as a blanking period), occurs during this period. This delay must be accounted for when emulating the video hardware to ensure timing remains accurate.

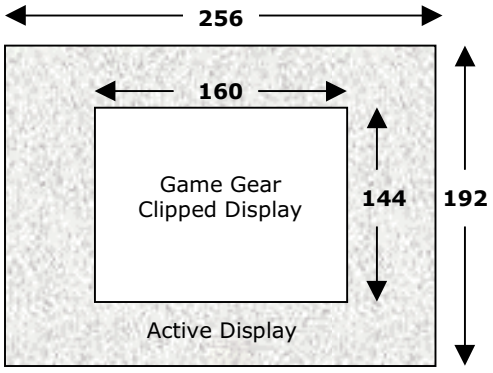


Figure 3.14 – Game Gear Display

The Game Gear hardware has its own in-built display, which operates using NTSC timing. The active display, however, is clipped to a 160x144 window, as shown in Figure 3.14. The entire active display is generated by the VDP, but is not visible.

The timing data can be used to establish the number of clock cycles the CPU emulation should execute per line and per frame, with the following formula:

$$\text{Clock Cycles Per Line} = \frac{\text{Clock Speed (Hz)} / \text{Scanlines Per Frame}}{\text{Frames Per Second}}$$

Table 3.14 shows the result of applying this formula to both the NTSC and PAL standards.

	Clock Speed (Mhz)	Clock Speed (hz)	Scanlines Per Frame	Frames Per Second	Clock Cycles Per Line
NTSC	3.58MHz	3,580,000	262	60	228
PAL	3.58MHz	3,580,000	312	50	230

Table 3.14 – Clock cycles required for each line rendered

This highlights the importance of accurate CPU timing, as discussed in section 3.2.2 (p.18). The correct number of instructions should be executed in a period of 228 or 230 clock cycles for each scanline rendered.

Horizontal and Vertical Ports

The horizontal and vertical ports can be read by software to establish the current position of the electron beam on-screen. The vertical port returns a value between 0 and 255, corresponding to a line of the display. The vertical port caters for frames with either 262 or 312 scanlines, values greater than the maximum it can store. The value does not wrap when it exceeds its limit; instead it jumps back to a particular value. The values the port jumps from and to are dependent on the screen mode in use. For example, in NTSC 256x192 mode, the port reaches value 218 and jumps back to 213, preventing an overflow beyond 255.

The horizontal port is only read by games making using of the lightgun peripheral; for this reason, it won't be analysed, as lightgun emulation will not feature in JavaGear.

3.4 – SN76489 Programmable Sound Generator

The Master System contains a Texas Instruments SN76489 Programmable Sound Generator (PSG). The PSG is integrated with the VDP chip (Figure 3.4), and is programmed through the Z80's I/O ports in a similar manner. Four audio channels can be simultaneously controlled and mixed, comprising three tone generators and one noise generator.

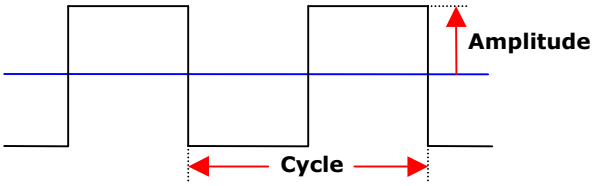


Figure 3.15 – Square Waves

The tone generators produce square waves at varying amplitudes and frequencies (Figure 3.15). The amplitude of the wave is measured by its distance from the central line. The greater the difference, the louder the sound. The frequency of the sound is measured by the rate at which it vibrates, the number of cycles per second. A cycle represents a full movement between positive and negative amplitudes. The greater the number, the higher the pitch.

The noise generator produces white noise and periodic (or synchronous) noise. It is typically used for sound effects and drum sounds in music. Noise is generated by a noise source, specified in the PSG data sheet [15] as, "a shift register with an exclusive OR feedback network." The precise nature of this feedback network is unknown, but tests have produced networks that sound similar [16]. The frequency of the noise channel specifies the rate at which to adjust the shift register. Once per cycle it is shifted right by one bit, if bit 0 is set the XOR feedback network is used to adjust the register. The noise generator does not produce a wave with a negative amplitude like the tone generators; instead the amplitude varies between zero and a positive value.

Writing two bytes to either port 0x7E or 0x7F programs the channels. This byte sets one of eight internal PSG registers that control a specific channel's properties. Unlike the VDP, the PSG does not contain an internal flag to record the byte due to be written. The byte itself specifies which of the two it is; if bit 7 is set, it is the first byte, if reset, it is the second byte.

The volume of a channel is set by the first byte (Table 3.15).

	7	6	5	4	3	2	1	0
Byte 1	1	C1	C0	1	V3	V2	V1	V0

Table 3.15 – Setting Channel Volume

Bits C0 and C1 select the channel to control, where the tone generators are mapped to the first 3 channels (0-2) and the noise generator is mapped to the final channel (3). V0-V3 contain the actual volume, a value between 0 and 15 where 15 represents silence.

The frequency of a tone channel is set by a combination of the two bytes Table 3.16).

	7	6	5	4	3	2	1	0
Byte 1	1	C1	C0	0	F3	F2	F1	F0
Byte 2	0	-	F9	F8	F7	F6	F5	F4

Table 3.16 – Setting Tone Generator Frequency

The main difference between the first byte when setting frequency as opposed to volume is that bit 4 is reset. This bit controls the operation taking place. Ten bits (F0–F9) define half the period of the desired frequency, a value between 0 and

0x3FF where 0 represents the highest pitch. The second byte can be rewritten, without rewriting the first.

The frequency of the noise channel is set by the first byte only (Table 3.17).

	7	6	5	4	3	2	1	0
Byte 1	1	1	1	0	-	Feedback	F1	F0

Table 3.17 – Setting Noise Generator Frequency

The feedback bit specifies the type of noise to generate. White noise is generated when set and periodic noise when reset. The frequency of the noise is controlled by two bits (F0-F1) as opposed to ten. Table 3.18 illustrates their function:

F0	F1	Operation Performed
0	0	High Pitch Noise
1	0	Medium Pitch Noise
0	1	Low Pitch Noise
1	1	Use Frequency of Tone Generator 3

Table 3.18 – Noise Generator Frequencies

The final sound is output by an amplifier summing circuit, which sums the tone generator and noise generator outputs.

3.5 – Cartridges and Memory Mapping

Master System and Game Gear software retails in cartridge format. This format offers the practicality of being robust, with no moving parts, making it suitable for use by children. The code is stored on a ROM chip, eliminating loading delays, allowing gameplay to be fast and continuous. In addition, distributing the software in this manner makes it less susceptible to piracy by an average user. Figures 3.16 and 3.17 show the outer casing of a Master System and Game Gear cartridge [17].

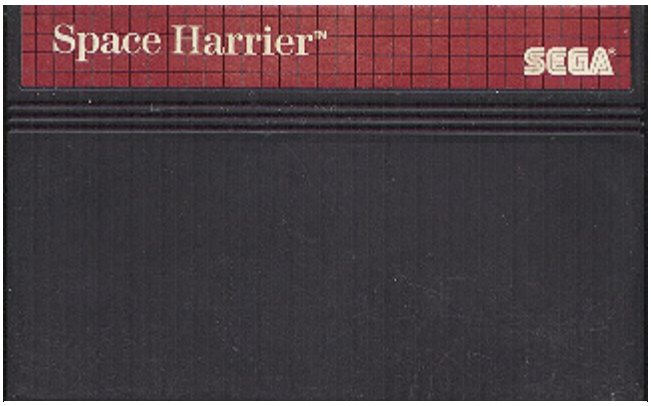


Figure 3.16 – Space Harrier for Master System

Figure 3.17 –
Space Harrier for Game Gear

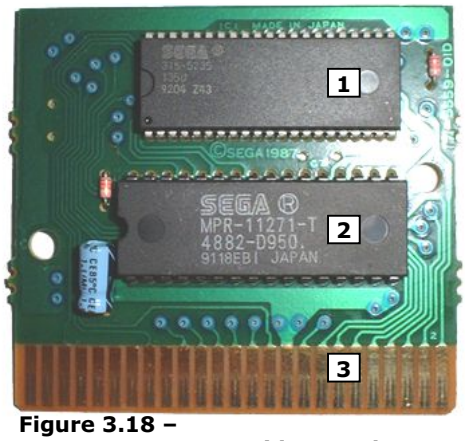


Figure 3.18 –
Master System Cartridge Hardware

Key

1	Memory Mapper Chip
2	Program ROM
3	Edge Connector

Master System and Game Gear cartridges contain ROM chips of various sizes; 32k, 64k, 128k, 256k, 512k and 1 megabyte. The Z80, however, can only address 64K of memory simultaneously, which is mapped between RAM and the cartridge ROM. The smallest cartridges can be directly mapped to the Z80's address space, but the larger cartridges require paging hardware to be mapped into the address space.

As Figure 3.18 [18] illustrates, cartridges containing a ROM greater than 32K include an additional memory mapping chip, which controls code paging to the appropriate memory frame. The system's memory map (Figure 3.19) comprises three frames and the system's RAM. Pages are mapped to frames by writing a page number to the appropriate frame control register located between memory locations 0xFFFD and 0xFFFF. The first 1K of frame 0 is not paged with the remainder of the frame; it contains system initialisation code and interrupt handling routines fixed at set locations of memory.

Cartridges may also contain on-board RAM to complement the system's existing 8K of memory. Some of these cartridges are equipped with a battery, allowing its contents to be preserved when the system is powered down. This functionality is used to save games, so that play can be continued at a later date. Frame 2 may be mapped to either cartridge RAM or ROM by writing to memory location 0xFFFC (Table 3.19). Bit 3 selects whether to page the frame with RAM or ROM, and bit 2 selects between one of the two pages of cartridge RAM.

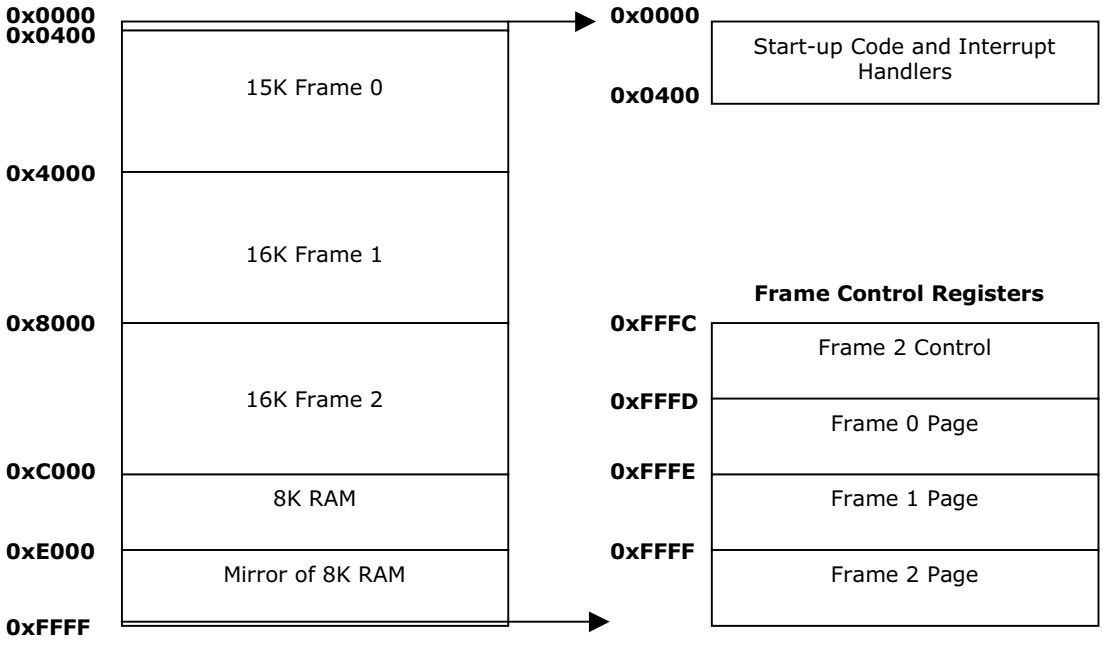


Figure 3.19 – Memory Map

7	6	5	4	3	2	1	0
-	-	-	-	ROM/RAM Select	Use RAM Page 0/1	-	-

Table 3.19 – 0xFFFC Frame 2 Control Byte

3.5.1 - Reading Cartridges

Figure 3.20 - SMSReader

Cartridges can be 'dumped' into a binary file on a computer using a homemade cartridge reader, like SMSReader (Figure 3.20). The schematics for such readers are freely available on hobbyist websites [18].

Other websites exist that openly post commercial ROM images for download. This is a simpler way of acquiring cartridge 'dumps', but the legality of such sites is questionable, they have no means of ensuring that users download titles they have originally purchased in cartridge format.

There is also a small range of software created by hobbyist coders and released into the public domain. This software could be distributed with the emulator for testing purposes.

4 DESIGN

4.1 – Design Overview

It may appear that freedom, when designing an emulator, is limited by the specifications of the original hardware. This is not true, there are multiple ways of emulating each component, as this report seeks to prove. The need to code various fast and accurate algorithms that function in co-ordination makes project success highly dependent on meticulous design from the onset.

It makes sense to implement a modular design where each hardware component is self-contained. This ensures the class structure is logical to someone familiar with the original design, i.e. the Z80 class emulates the Z80 processor. This design means that each component can be reused when emulating a different machine with similar hardware. For example, the Z80 class could be reused in a project to emulate the Z80 based Pac-Man arcade machine, and the SN76496 class could be reused in a project to emulate a BBC Micro computer, as it contains the same sound processor.

4.2 – Z80 Class

A variety of methods can be used to emulate a CPU. The most common is interpretative emulation. Its popularity resides in the fact that its algorithm (Figure 4.1) is identical to the fetch, decode and execute loop of a real CPU. This means debugging the emulation code is relatively simple. Opcodes can be timed accurately, and interrupts are easy to perform. The disadvantage with this method is that it is slow.

```
while (cpu_running)
{
  // Fetch Opcode from Memory
  // Decode Opcode
  // Execute Opcode
}
```

Figure 4.1 – Interpretative Emulation Algorithm

```
while (cpu_running)
{
  // Identify Section of Code
  if (block_cached)
    // Execute Block
  else
  {
    // Generate Native Code
    // Execute Block
  }
}
```

Figure 4.2 – Dynamic Recompilation Algorithm

Dynamic recompilation differs from interpretative emulation in that instead of decoding single instructions, multiple blocks of instructions are decoded and cached (Figure 4.2). This makes subsequent execution of program blocks much faster, as they have been compiled to native code. A block of code is usually defined as the set of instructions before a CALL or JUMP is made. The main drawback with this method is that it greatly increases the complexity of the CPU core. Accuracy is reduced as pending interrupts are checked for at the end of each block as opposed to after every instruction.

The author of TARMAC, a dynamically recompiling ARM emulator, remarks with regard to tackling a similar project that, "debugging a dynamic recompiler is a job for Sherlock Holmes … [it] pushes the boundaries of what is feasible in the time available for a university project. [19]" JavaGear aims to emulate a complete system, rather than a single processor, making dynamic recompilation unfeasible for this project. Most importantly, dynamic recompilation would not be possible in Java, because of the multiple host platforms on which the JVM runs and security restrictions in creating native code.

A fast algorithm operable on Java is threaded emulation (not to be confused with Java Threads). Threaded emulation is similar to dynamic recompilation, but instead of generating native code for blocks of instructions, a list of addresses to existing subroutines is generated. This means that the fetch and decode operations only occur at the start of a block, as opposed to before every instruction, which provides a speed increase. Timing is more accurate than dynamic recompilation, as execution of the list of addresses can be suspended and resumed as necessary. Unfortunately, this algorithm would not yield a speed increase when emulating the Z80. The decode stage of the Z80 is minimal, each opcode is mapped directly to a hex value, unlike certain CPUs (e.g. the ARM processors) where multiple manipulations must be performed to decode the hex value. Threaded emulation is theoretically possible, but not a practical option for Z80 emulation.

JavaGear uses an interpretative CPU core, as the faster algorithms are not appropriate when emulating the Z80 in Java. This increases the importance of code efficiency; the Z80 class will be executing around 500,000 emulated instructions a second, so even small changes to the design could have dramatic consequences to its speed. Figure 4.3 illustrates the algorithm deployed, it allows the CPU core to execute a specific number of machine cycles.

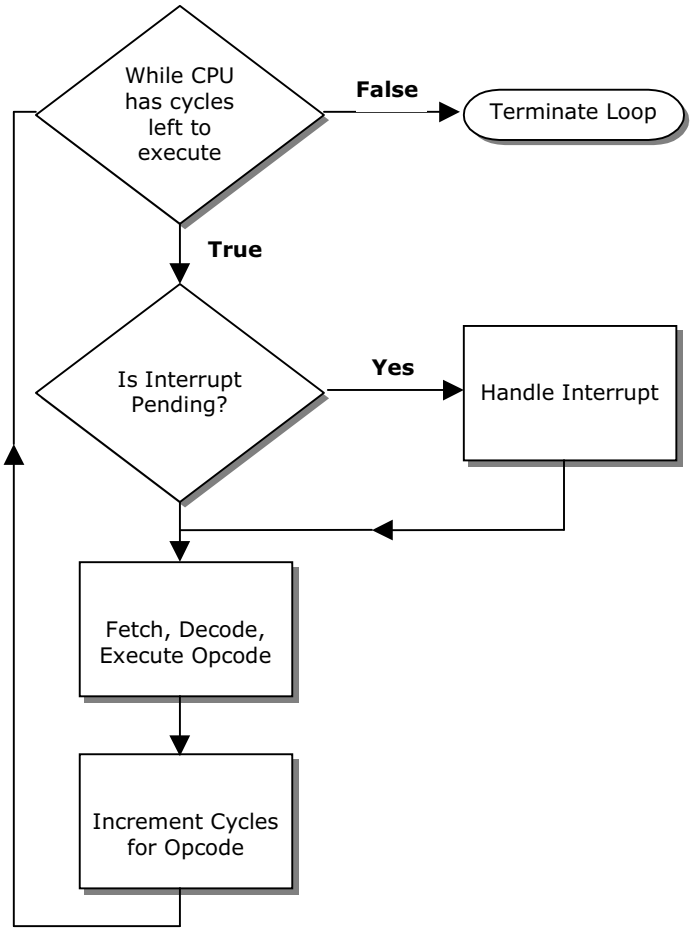


Figure 4.3 – Z80 Class Interpretative Algorithm

4.3 – Memory Class

The Memory class handles a variety of functions; firstly, it emulates the system's main memory map; secondly, it emulates the cartridge's paging hardware and thirdly, it handles reading a ROM file from disk and configuring the frames and pages appropriately. It should be noted that the Memory class does not contain high-level GUI code to handle file selection, it only contains the low-level configuration code.

When JavaGear runs in applet mode, ROM files will be fetched from a remote server. Master System titles can be up to 1024k in size, which means that with a 56K modem they will take around 5 minutes to download. This is unacceptable; the Memory class has the ability to read files compressed using ZIP format. The ZIP format typically reduces the size of a Master System ROM by 40-60%. This greatly reduces loading time when using a slow connection.

4.4 – Ports Class

The Ports class is used to connect devices (i.e. Java objects) to the Z80's ports. Devices can be connected to the in port and/or out port. They are then addressed in an identical manner to the original hardware.

4.5 – Controllers Class

The Controllers class converts input from a PC keyboard into a format representing the state of the Master System control pads. It allows a PC keyboard to be used in place of the original controllers. It is attached to the Ports class.

4.6 – VDP Class

The VDP class includes emulation of all VDP functionality, comprising control and data ports, interrupts and display layer generation. It creates an array of pixels, representing a single frame of the Master System display. It should be noted that the Java routines to output this array to the screen are abstracted to a separate class; the VDP class does not perform any drawing itself, it simply stores a representation of the display. This provides the advantage of separating VDP emulation from Java specific rendering code, enhancing code readability.

The display is emulated using a line by line algorithm. A procedure is called to generate a particular line of the display. This is a slower, but more accurate mechanism than generating the entire display simultaneously. It supports changes to the screen that occur between a particular scanline. Many titles, including *Sonic the Hedgehog*, update the palette between scanlines; only the line by line algorithm is capable of emulating this behaviour. In addition, line interrupts can be generated in an identical manner to the original hardware, so graphical effects are supported.

Figure 4.4 provides a basic overview of the line rendering algorithm, illustrating how priorities are preserved between the background and sprite layers. An array is used to record background tile pixels of a higher priority than the sprite layer.

After each line of the display has been generated, an interrupt procedure is called to establish whether the Z80's interrupt line should be asserted (Figure 4.5). This procedure only asserts the line; it is cleared by reading the control port and by adjusting various VDP registers.

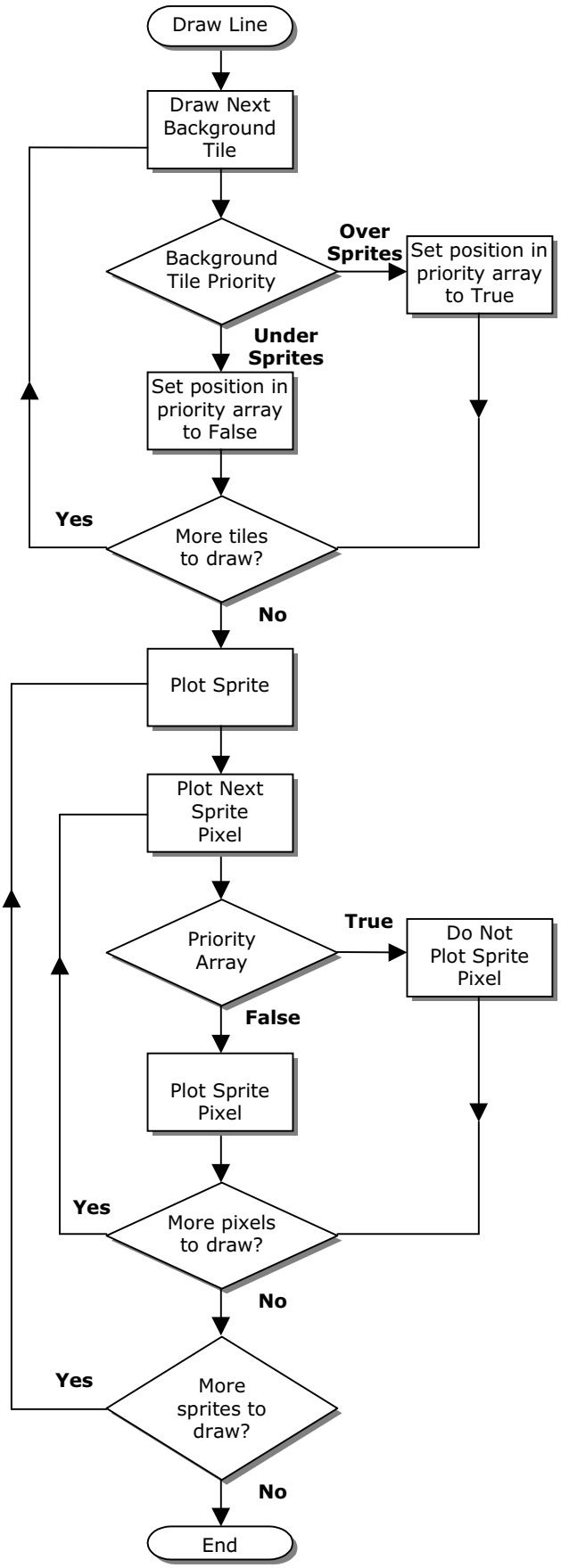


Figure 4.4 – VDP Line Rendering Algorithm

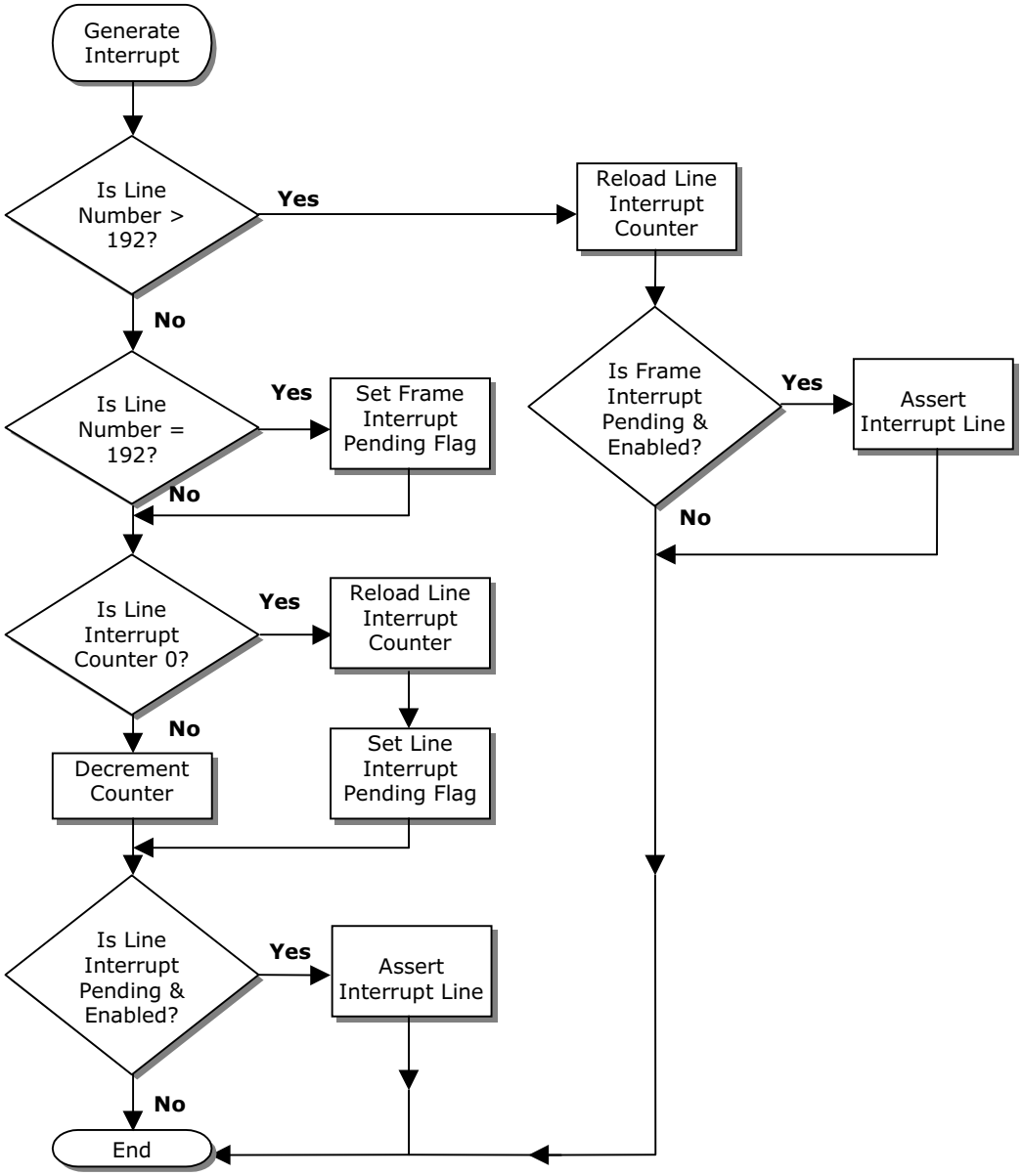


Figure 4.5 – VDP Interrupt Generation Algorithm

The VDP emulation has the ability to support both NTSC and PAL timing standards. Certain titles were programmed with a particular variant of the Master System in mind, and are dependent on exact timings. For example, *Back to the Future 3* only works on PAL systems, many emulators do not support this title because PAL timing is not offered.

Certain VDP functionality was omitted due to time constraints. As detailed, the Master System VDP is a customised TMS9918 VDP. It is possible for software to use legacy screen modes derived from the TMS9918. This is rare, and only one title, *F16 Fighter*, appears to use such a screen mode. It was deemed an inappropriate use of time to implement a new rendering algorithm for just one title and development was focused on perfecting the standard screen mode.

4.7 – SN76496 Class

The SN76496 emulation consists of three main classes, the SN76496 class, the ToneGenerator class and the NoiseGenerator class. Its design mirrors the specification of the original chip, three independent tone generator instances and one noise generator instance are created. Each generator outputs a signed value, representative of its amplitude. Finally, the values are summed to mix the channels (Figure 4.6). An appropriate number of amplitude values are generated for the period of time emulated. The design of the class synthesises and mixes the sound in real-time. This ensures faithful reproduction of effects like vibrato that rapidly alter the shape of the waveform.

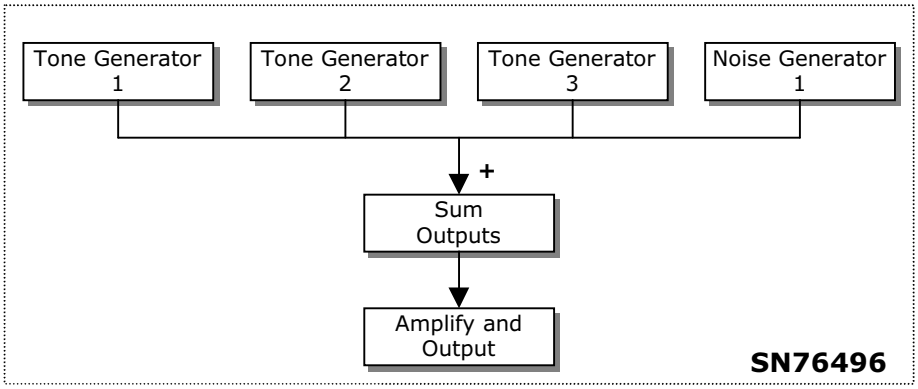


Figure 4.6 – SN76496 Class

Figure 4.7 demonstrates the algorithm used by each tone generator instance to produce a square wave at a variable frequency. A counter loaded with half the frequency value, decrements each time the algorithm is called. This counter stores the period of time until the wave flips between a positive and negative value. An amplitude flip-flop is toggled to determine the current state. Once flipped twice, a full cycle of sound is generated. The values of the counter and flip-flop are preserved between each algorithm call.

This algorithm is most appropriately explained by following a simple example (Figure 4.8). Initially, the counter is loaded with value 5, although in practice it is likely to be a far greater value. The rate at which the counter decrements is determined by the clock speed of the PSG chip and the sample rate at which the emulated sound is produced. For simplicity, the rate will be fixed at 1 for this example. The table shows the state of the counter and amplitude flip-flop *after* the algorithm is called.

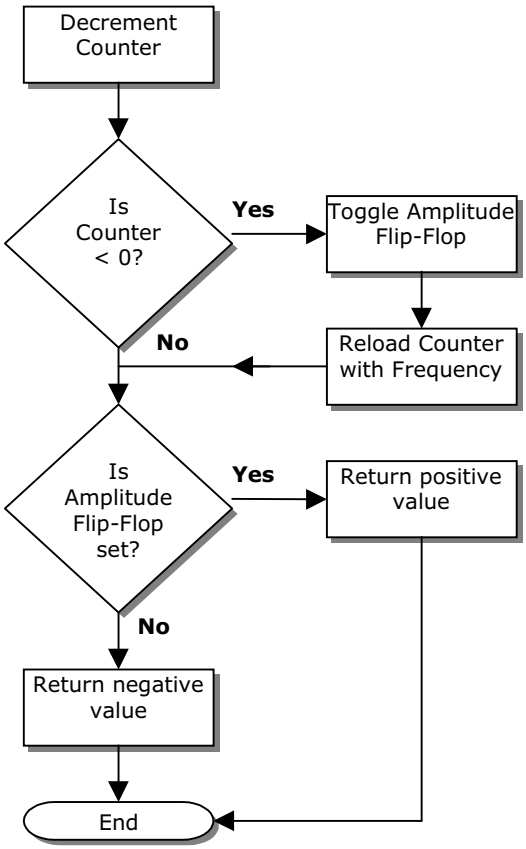


Figure 4.7 – Square Wave Generation

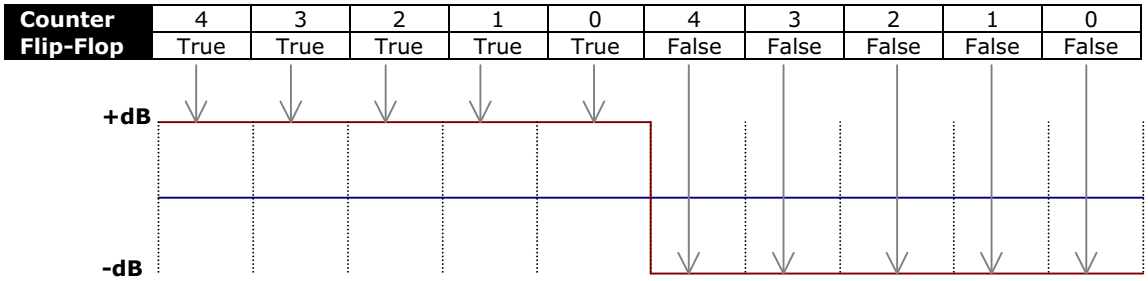


Figure 4.8 – Square Wave Algorithm Example

Noise generation uses a more complex algorithm, but the underlying structure is similar, so its design will not be detailed here.

4.8 – EmulateLoop Class

The EmulateLoop class synchronises the emulation components to provide a persistent emulation of the system. In execution, the class creates the impression that the components are running in parallel, as they do on the original hardware. In fact, each component is run sequentially for a set period of time. The original design may suggest that the components would be best implemented as a series of threads run simultaneously. However, co-ordination between each component would be difficult, reducing the overall emulation speed. Running the components sequentially guarantees both accurate timing and speed advantages.

Figure 4.9 encompasses the interpretative Z80 algorithm (Figure 4.3), VDP line rendering algorithm (Figure 4.4), VDP interrupt generation algorithm (Figure 4.5) and sound output algorithm (Figure 4.6). It shows the interdependencies of the classes outlined in the design of JavaGear.

The emulation loop is highly processor intensive. It is implemented as a thread to ensure that it can be effectively time-sliced by the JVM. This allows other aspects of JavaGear to be simultaneously updated and refreshed (e.g. other windows and events).

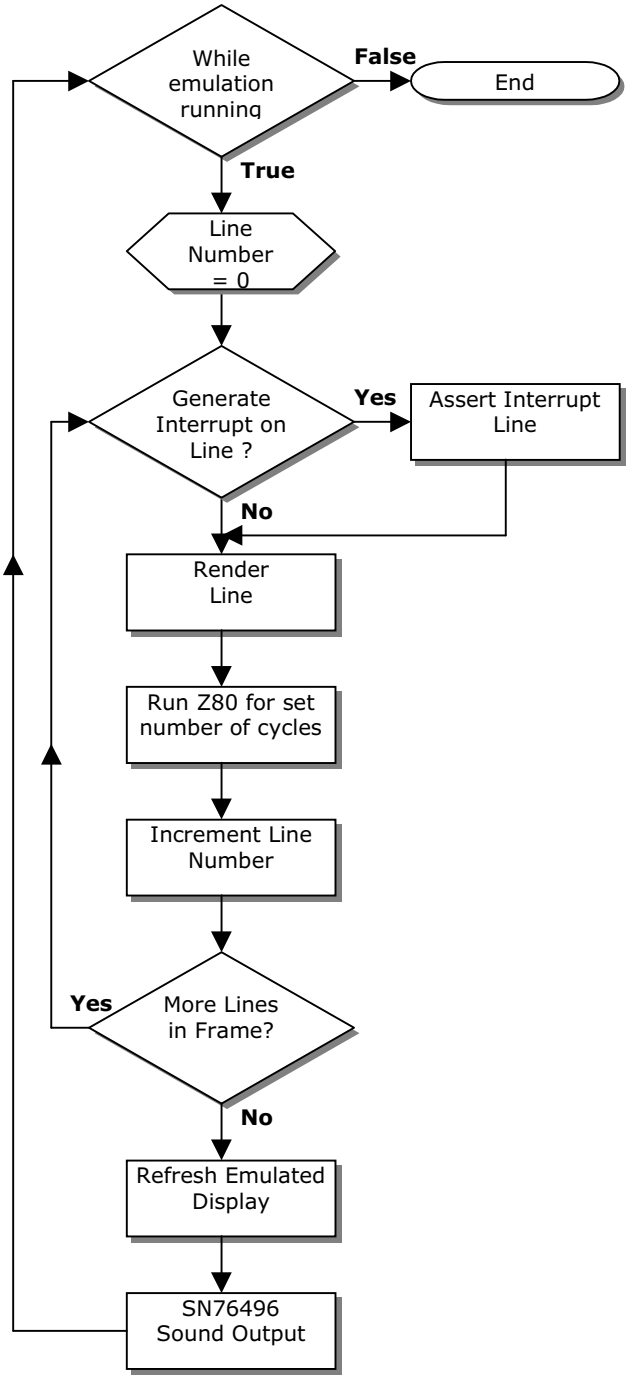


Figure 4.9 – Main Emulation Loop

The time taken to generate the frame and output the sound is recorded and contrasted with the exact timings of the original hardware. If the host platform generates a frame in a shorter period of time, the generation of the subsequent frame is delayed appropriately. This locks emulation speed to the timings dictated by the NTSC and PAL standards. (This timing is not to be confused with the opcode emulation timing present in the CPU core.)

4.9 – Class Diagram

The following UML diagram (Figure 4.10) illustrates the associations between the classes discussed in this section. Explicit details about the individual classes and the data structures used can be found in the implementation section (p.44).

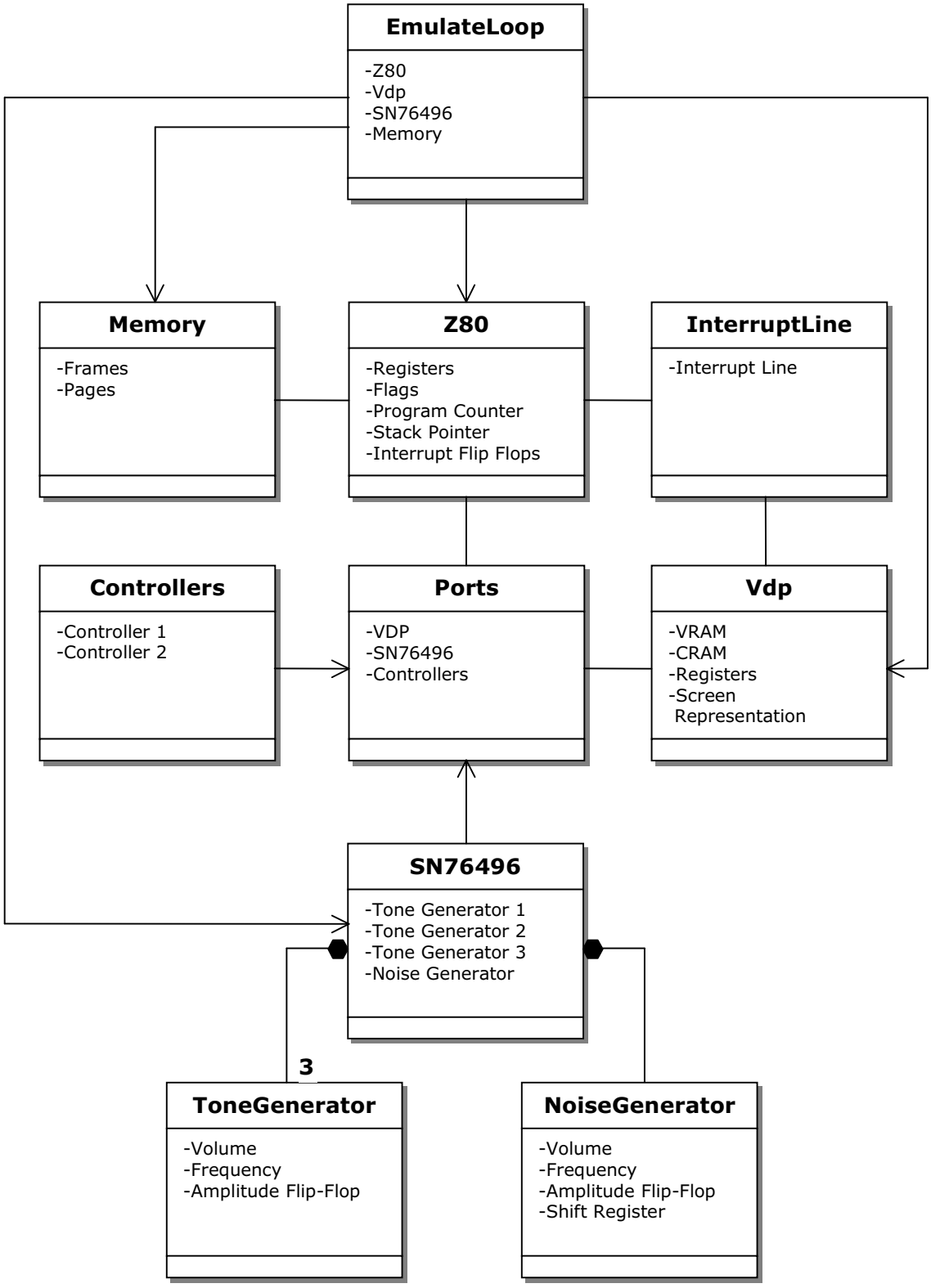


Figure 4.10 – Class Diagram

5 PROTOTYPES

Designing and implementing an emulator is a complex process; the interdependencies and complexities of the components force a lengthy development period with few visible results. The most intricate component, the CPU, must be successfully emulated before work on the other components can commence. JavaGear is a project with a multitude of risk factors (Section 8 p.59), all with the capability to derail the entire project, compromising not just its success but its ability to function at all.

Risk was minimised by verifying the design (Section 4 p.32) of the essential components with an initial prototype capable of executing a single Master System binary. Evolutionary prototyping ensured knowledge ascertained in the early stages of development could be refined through iterations and that the technology was proven and robust. In this section, only the first prototype will be detailed; it demonstrates the way in which a complex project with interdependent components can be initiated with immediate results.

5.1 – Aims

The prototype will successfully run a simple Master System binary entitled 'Only Words'. [20] The prototype will verify the information collected during background research is correct (Section 2 p.9), and that it has been successfully understood. It will ensure the design (Section 4 p.32) is sound and provide an indication of the time the full implementation is likely to take. It will minimise the skills and technological risks inherent in a project of this nature.

5.2 – Specifications

As its title suggests, Only Words is a simple demo capable of displaying text to the screen. It was developed by an amateur programmer to illustrate the basics of Master System programming. The advantage of performing an initial implementation that executes this title is that, unlike commercial titles, a copy of the source code is provided. Debugging commercial titles, even with a competent knowledge of Z80 assembly language, is extremely complex. Comprehending the code routines is reliant on experimentation and guesswork. The source code provided with Only Words is well documented, which reduced the time spent debugging the emulation code.

Z80	✓
… Opcodes (40 of 1,268)	✓
… Interrupts	✗
Memory	✓
… ROM Loader	✓
… Memory Mapper	✗
Ports	✓
VDP	✓
… Control and Data Ports	✓
… Interrupts	✗
… Tile Decoding	✓
… Background Layer	✓
… Horizontal / Vertical Scrolling	✗
… Sprite Layer	✗
Controllers	✗
SN76496	✗

Figure 5.1 – Emulated Prototype Components

Referral to the source code verified the hardware components and features used by the demo (Figure 5.1). Unusually, Only Words is not reliant on any form of Z80 or VDP interrupts to run successfully. This simplifies the emulation process considerably, as it is not dependent on any form of timing; the CPU can be run until the demo code loops or finishes and finally the contents of VRAM displayed.

5.3 – Implementation

The Z80 implementation was limited to the opcodes required by the demo. Each time a new opcode was implemented, the code was recompiled and the output verified against the source code. The internal properties of the Z80 are printed to the console after executing each opcode (Figure 5.2). This proved an invaluable way of correcting faulty opcode interpretations and other bugs.

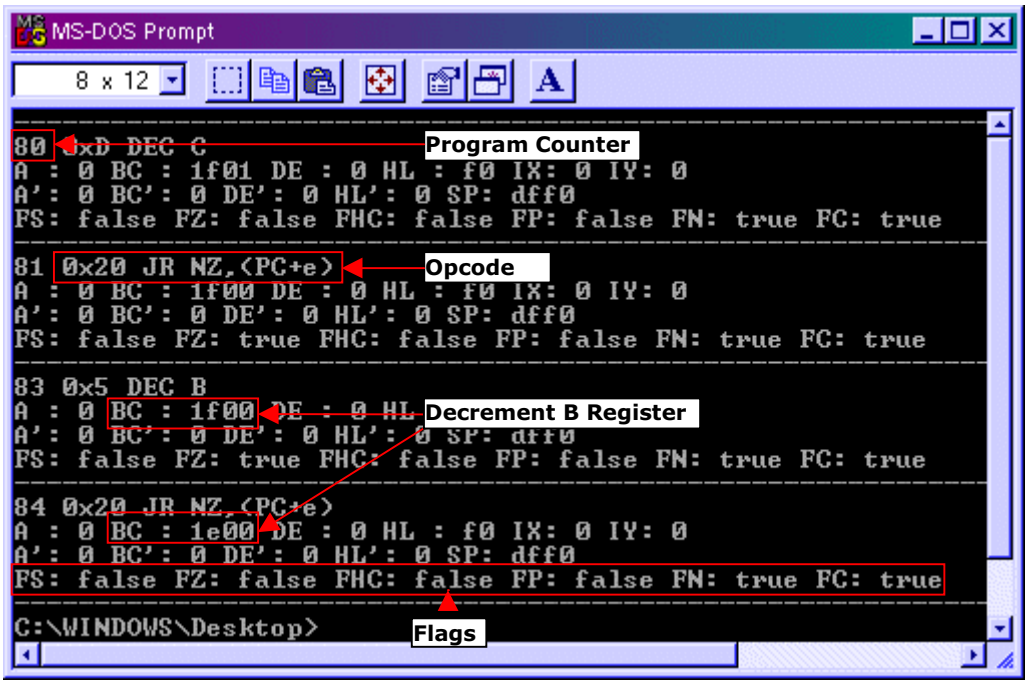


Figure 5.2 – Z80 Console Output

The graphical output from this initial prototype differs considerably from the final product. The prototype is a low-level implementation of the components and has no GUI, or Java drawing routines. Instead, a far simpler mechanism was developed to verify the accuracy of the VDP emulation. The background layer is rendered by producing a text file containing 256 characters on 192 lines. The demo only uses black and white graphics, so a black pixel can be represented by writing a hash symbol '**#**' and a white pixel by writing a space ' ' (Figure 5.4). The output can be displayed in a text editor, using a fixed width font (Figure 5.3).

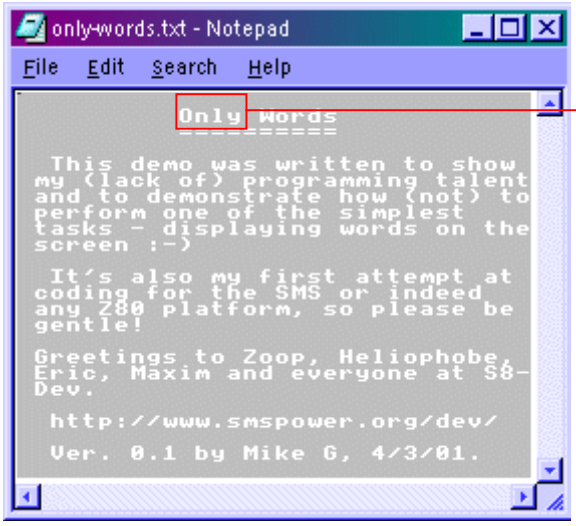


Figure 5.3 – Text Output from Prototype
Font: Courier New (Size 1)

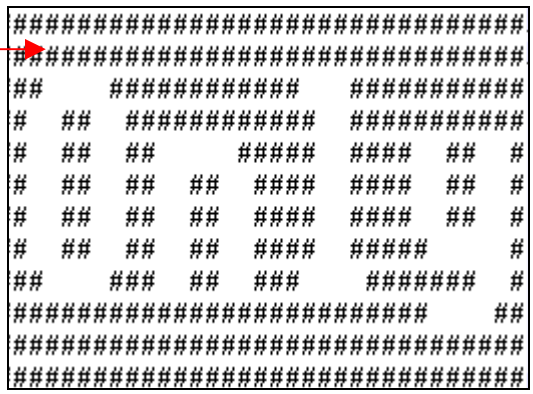


Figure 5.4 – Close Up Of Text Output
Font: Courier New (Size 10)

To clarify, fonts on the Master System are composed of tiles in the same manner as sprites and other graphics. The output can be compared with that from a real Master System console (Figure 5.5). The green area surrounding the active display is the outer border first illustrated in Figure 3.13. It serves no particular purpose, and is not emulated. The text background is darker on the Master System output because it consists of pure black pixels rather than hash symbols.

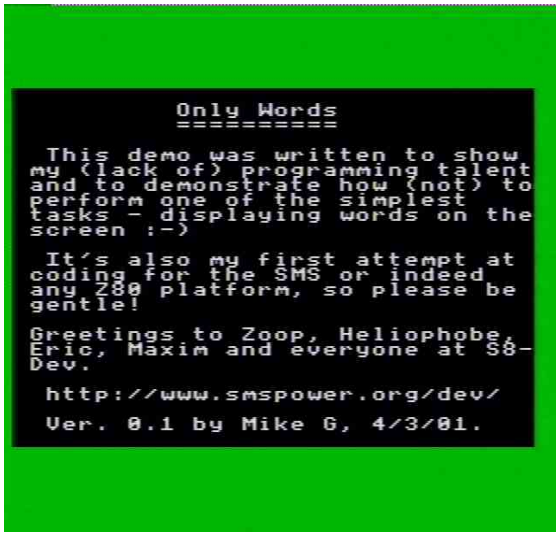


Figure 5.5 – Output From Real Master System

Debugging the VDP in this manner proved to be effective. For example, the first iteration of the tile decoding algorithm produced an incorrect mirrored output. The rudimentary output allowed the initial VDP code to be perfected without coding Java rendering routines.

5.4 – Conclusions

Prototyping proved invaluable in achieving instant visible results. Problems that arose could be tackled from the onset, reducing later debugging work. "The biggest technological risks are inherent in how the components of a design fit together rather than being present in any of the components themselves. [21]" The prototype minimised this technological risk by producing a working, integrated model of the components. Skills risks were assessed, further background research undertaken as appropriate and the project's validity established.

6 IMPLEMENTATION

6.1 – Implementation Overview

A bottom-up approach was adopted for the implementation process; this concentrated early development in high-risk areas where failure would compromise the project (e.g. the CPU and memory emulation). High level components, including the GUI and controller configuration functionality, were among the last features to be implemented; they did not represent a significant design risk and were likely to evolve as the project progressed.

The code in this section should be studied in conjunction with the information ascertained during background research (Section 3, p.15). A direct comparison can then be made between hardware and software, verifying the accuracy of emulation.

6.2 – Z80 CPU

The Z80's registers are represented by Java's 32-bit integer primitive. This may appear to be an unconventional choice as the registers store either 8 or 16-bit values. It would appear logical to represent the registers using 8-bit byte and 16-bit char primitives. This would be problematic for a number of reasons. Firstly, the Java byte primitive is a signed type, whereas the Z80's registers are unsigned. This would lead to erroneous arithmetic operations as Java promotes types of integer and smaller to integers before binary operations [22]. Secondly, Java does not have built-in overflow detection, and therefore a manual test needs to be performed after an operation to establish whether the register is holding a value greater than its designated size. If a 16-bit char primitive represented a 16-bit register, there would be no way to detect whether overflow had occurred during an operation. For this reason, the primitive must be capable of holding a value greater than the emulated register. Thirdly, int is Java's fastest data type.

```
class Registers
{
  // First Bank
  int high, low;
  // Second Bank
  int high2, low2;

  // Set 16-Bit Pair
  public void set(int value)
  {
    low  = value & 0xFF;
    high = (value>>8) & 0xFF;
  }
  // Set Lower 8-Bits
  public void setL(int value) {……}
  // Set Higher 8-Bits
  public void setH(int value) {……}
  // Get 16-Bit Pair
  public int get()
  {
     return (high<<8) | low;
  }
  // Get Lower 8-Bits
  public int getL() {……}
  // Get Higher 8-Bits
  public int getH() {……}
  // Swap Banks
  public void ex() {……}
}
```

Figure 6.1 – Registers Class

An instance of the Registers class can emulate either the BC, DE, HL, IX or IY register pair (Figure 6.1). This class provides the functionality to access the register pair as a single 16-bit value, or as two 8-bit values. Each instance supports both banks of the register pair, (e.g. BC and BC'). The banks can be swapped by calling the ex() procedure. The Registers class ensures code readability when implementing the opcodes, avoiding the excessive use of bitwise operators.

A unique Accumulator class exists to emulate the accumulator. The accumulator is affected differently by many operations to the other register pairs, and benefits from an independent class with customised methods and procedures. Similarly, there is a unique Refresh and Flag class.

The analysis of CPU algorithms (Section 4.2, p.32) concluded that an interpretative core was most appropriate. The opcode decoding algorithm of such a core can be implemented in a number of ways. For example, an increment operation is encoded as the following byte:

7	6	5	4	3	2	1	0
0	0	**r**	**r**	**r**	1	0	0

Table 6.1 – Encoded Increment Operation

The value of 'r' specifies the register to increment. It is decoded as follows:

Register	r	Register	r
A	111	E	011
B	000	H	100
C	001	L	101
D	010		

Table 6.2 – Decoded meaning of 'r'

Each opcode could be decoded as part of a two-step procedure. Firstly, the actual operation could be decoded, followed by the particular operands. This method would minimise the code required, and is the most elegant approach. A limitation is the time spent in the decoding phase, which would greatly increase. Attempts to use this two-step technique in emulators have produced slow CPU cores that were difficult to debug [23]. Combined with Java's speed deficiencies, this method was deemed inappropriate if the CPU core was to decode 500,000 instructions a second on a modestly specified platform.

A faster, and more widely adopted approach, is to manually decode the opcodes before runtime. The increment operation (Figure 6.1) can be decoded into the following byte codes:

Operation	Byte Code	Operation	Byte Code
INC A	0x3C	INC E	0x1C
INC B	0x04	INC H	0x24
INC C	0x0C	INC L	0x2C
INC D	0x14		

Table 6.3 – Byte Codes for Increment Operation

The decode phase is now considerably faster; a byte in memory can be directly mapped to a particular operation. Figure 6.2 illustrates how this method was implemented in JavaGear.

```
// Decode and Execute Opcode
public void doOp(int opcode)
{
   switch(opcode)                                  // Switch decodes the opcode
   {
       case 0x04: bc.incH();      pc++; break;     // INC B
       case 0x0C: bc.incL();      pc++; break;     // INC C
       case 0x0D: de.incH();      pc++; break;     // INC D
       case 0x1C: de.incL();      pc++; break;     // INC E
       case 0x24: hl.incH();      pc++; break;     // INC H
       case 0x2C: hl.incL();      pc++; break;     // INC L
       case 0x3C: a.inc();        pc++; break;     // INC A
   }
}
```

Figure 6.2 – Opcode Decoding

This example only shows the increment operation. In reality, the opcode decoding procedure caters for all the Z80 operations. The switch procedure, which decodes

```
Method void doOp(int)
  0 aload_0
  1 getfield #30 <Field Refresh r>
  4 invokevirtual #104 <Method void inc()>
  7 iload_1
  8 tableswitch 0 to 255: default=7937
        0: 1048
        1: 1061
        2: 1094
        3: 1128
        4: 1148
        ……
        255: 7918
```

Figure 6.3 –
Opcode Decoding (VM Assembly Language)

the opcodes, is extremely efficient. Switch statements are compiled using either the *tableswitch* or *lookupswitch* instruction. The *tableswitch* instruction is the more efficient of the two, in terms of speed rather than space; the switch statement is indexed into a table, allowing rapid linear searches. To verify the *tableswitch* instruction was in use as opposed to the *lookupswitch* instruction, the Z80 class was decompiled with the Java Profiler to "virtual machine assembly language" (Figure 6.3).

The opcode decoding procedure is looped for a specified number of machine cycles (Figure 6.4). After an opcode is executed, the machine cycles are incremented appropriately. Between each instruction, the interrupt line is checked, and if asserted, the interrupt handling procedure called.

```
public void run(int cycles)
{
   tstates = 0;

   while (tstates < cycles)              // Run for a specific number of cycles
   {
      if (irq.getLine())                 // If Interrupt Line Asserted
        interrupt();                     // Call Interrupt Handling Procedure
      opcode = mem.read(pc);             // Fetch Opcode From Memory
      doOp(opcode);                      // Decode And Execute Opcode
      tstates += opstates[opcode];       // Increment TStates
   }
}
```

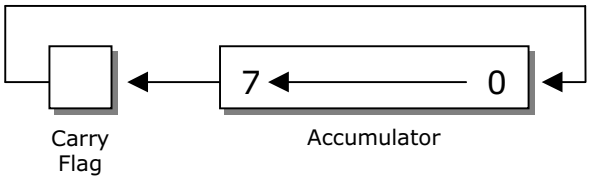
Figure 6.4 – Opcode Fetch and Decode Loop

Once the operation is decoded, the appropriate procedure is called to handle its emulation. The majority of these procedures are split between the Accumulator and Registers classes. Operations tend to affect the accumulator in a different manner to the register pairs. For example, the ADD instruction requires a separate implementation for the accumulator as it is an 8-bit register, and the flags are adjusted differently to the 16-bit ADD operation. In addition, many instructions are unique to the accumulator, so its abstraction to a separate class is a logical step. Figure 6.5 provides an example of one such instruction, which rotates the accumulator left through the carry flag.

This object-orientated design offers a further advantage; many of the Z80 instructions operate in a similar manner on multiple registers. Each instruction can be implemented once, as part of the Registers class, and executed independently with each instance. This minimises the code required to emulate the instruction set.

As the Z80 is an 8-bit processor, all 1,208 instructions cannot all be mapped to a single byte. Additional instructions are denoted by the prefixes CB, DD, FD or ED. When an extended instruction is encountered, the main decoding procedure passes execution to a secondary procedure containing decoding information for the particular opcode.

```
private void rla()
{
  // Make Backup Copy Of Register
  int temp = reg;
  // Shift Register Left By One Bit
  reg = (reg << 1) & 0xFF;
  // Move Carry Flag to Bit 0
  if (f.carry())
    reg |= 0x01;
  else
    reg &=~0x01;
  // Move Original Bit 7 to Carry Flag
  if ((temp & 0x80) == 0x80)
    f.carryOn();
  else
    f.carryOff();
  // Set Other Flags Appropriately
  f.negativeOff(); f.setHalfCarry(false);
}
```



	Before	After
Accumulator	00111111	01111111
Carry Flag	True	False

Figure 6.5 – Rotate Accumulator Left

The interrupt procedure is called between instructions, whenever the interrupt line is asserted (Figure 6.6). Once an interrupt is accepted, interrupts are disabled, the line is cleared and the current program counter pushed onto the stack.

```
private final void interrupt()
{
  // Interrupts Not Allowed: Interrupt Flip Flop Disabled
  if (!iff1) return;
  // Interrupts Not Allowed: Directly after EI instruction
  if (EI_inst) return;
  // Increment Refresh Register Appropriately
  r.inc();
  // Reset Interrupt Flip Flops
  iff1 = false;
  iff2 = false;
  // Clear Interrupt Line
  irq.setLine(false);
  // Interrupt Modes 0 and 1 have the same effect on Master System
  if ((im == 0) || (im == 1))
  {
    // Push Program Counter Onto Stack
    push(pc);
    // Set Program Counter to 0x38
    pc = 0x38;
    tstates+=13;
  }
}
```

Figure 6.6 – Interrupt Handling Procedure

The complete set of classes comprising the Z80 emulation are illustrated in Figure 6.7.

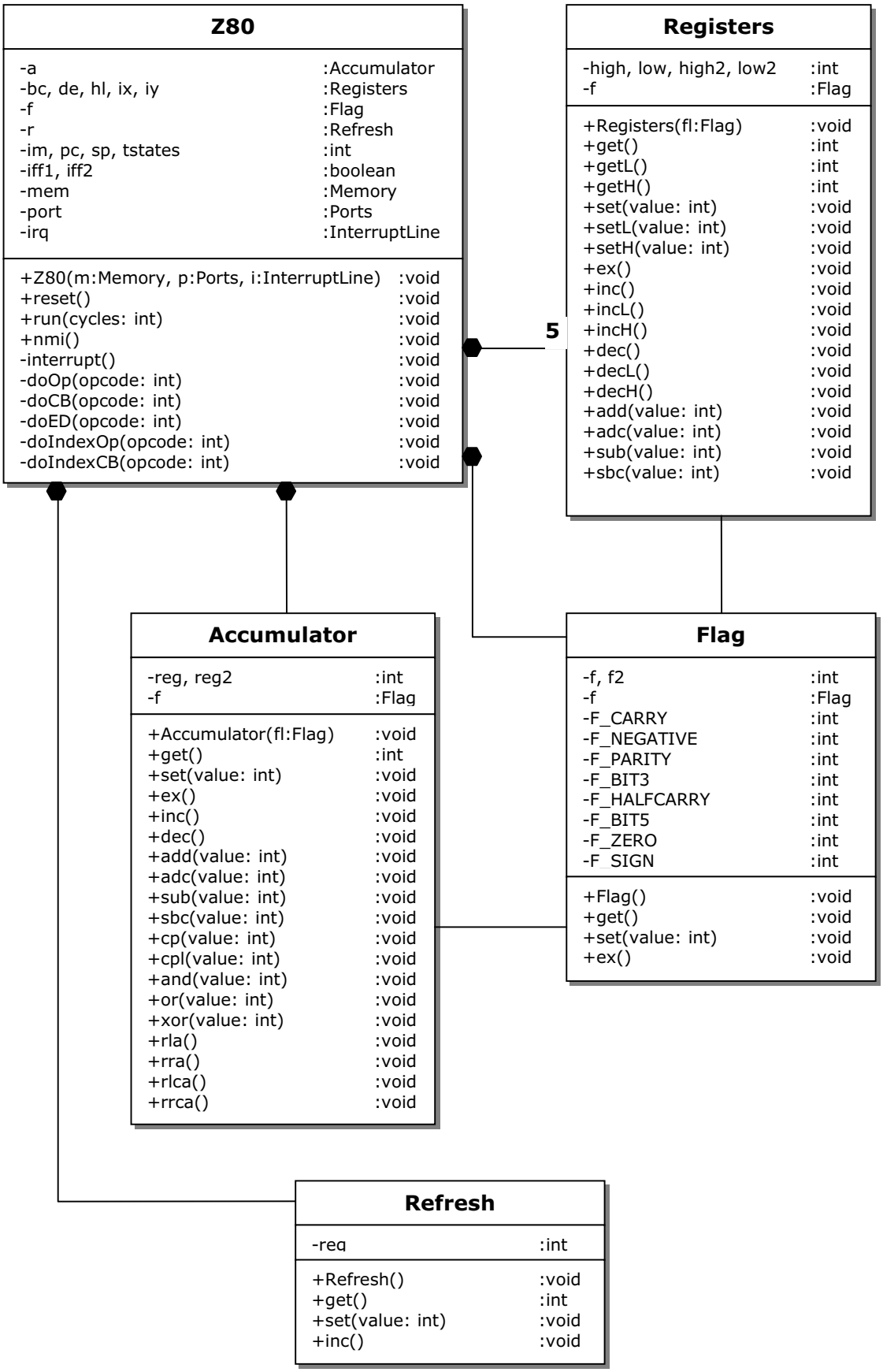


Figure 6.7 – Z80 Emulation UML Class Diagram

6.3 – Memory and Paging Emulation

The Master System's memory is represented by a series of two-dimensional arrays (Figure 6.8). The main array of frames, contains four 16K frames, which can be mapped to the ROM pages or cartridge RAM pages.

```
// Memory Frames (4x16K)
private byte [][] frames;
// ROM Pages (Number depends on size of ROM)
private byte [][] pages;
// Cartridge RAM Pages (2x16K)
private byte [][] cart_ram_pages;
```

Figure 6.8 – Memory Frames & Pages

```
// Setup Default Mapping
frames[0] = pages[0];
frames[1] = pages[1];
```

Figure 6.9 – Mapping Example

The use of two dimensional arrays ensures mapping is simple and fast (Figure 6.9). The frames array can be pointed at the appropriate page.

Figure 6.10 illustrates how a signed byte is read from memory. The virtual address is mapped to a physical address. The address is checked, and the appropriate frame used.

```
public int readSigned(int address)
{
  if (address < 0x400)                    // First 1K (Never Paged Out)
    return pages[0][ address];
  else if (address < 0x4000)              // ROM Page 0 (15K)
    return frames[0][address];
  else if (address < 0x8000)              // ROM Page 1 (16K)
    return frames[1][address-0x4000];
  else if (address < 0xC000)              // ROM Page 2 or Cart RAM (16K)
    return frames[2][address-0x8000];
  else if (address < 0xE000)              // RAM (8K)
    return frames[3][address-0xA000];
  else if (address < 0x10000)             // Mirror RAM (8K)
    return frames[3][address-0xC000];
  else                                    // Out of Bounds
    return 0;
}
```

Figure 6.10 – Reading a signed byte from memory

The paging hardware present in Master System cartridges is emulated by a paging procedure (Figure 6.11). When a memory location between 0xFFFC and 0xFFFF is written to, the procedure is called and the appropriate page mapped to memory.

```
private void page(int address, int value)
{
  switch(address)
  {
    // RAM/ROM Select Register
    case 0xFFFC:
      if ((value & 0x08) == 0x08)
      {  // Map RAM
        frame_two_rom = false;
        if ((value & 0x04) == 0x04)
        {  // Use Second RAM Page
          frames[2] = cart_ram_pages[1];
          cart_ram_page = 1;
        }
        else
        {  // Use First RAM Page
          frames[2] = cart_ram_pages[0];
          cart_ram_page = 0;
        }
      }
      else   // Map ROM
        frame_two_rom = true;
     break;
    // Frame 0 Page
    case 0xFFFD:
        frames[0] =
        pages[value%number_of_pages];
        break;
    // Frame 1 Page
    case 0xFFFE:
        frames[1] =
        pages[value%number_of_pages];
        break;
    // Frame 2 Page (ROM/RAM)
    case 0xFFFF:
        if (frame_two_rom)          // Map ROM
          frames[2] =
          pages[value%number_of_pages];
        else                        // Map RAM
          frames[2] = pages[cart_ram_page];
        break;
  }
  // Write value back to memory
  frames[3][address-0xC000]= (byte) value;
}
```

Figure 6.11 – Paging Procedure

6.4 – Video Emulation

The video display emulation consists of three key sets of procedures. The first set emulates the interface between the Z80 and the VDP, including the data port, control port and interrupt mechanisms. The second set generates an array of pixels containing a representation of the display. The third set, abstracted to a separate class for clarity, displays the pixel array to the screen.

The first stage of emulation ensures the VDP's main data structures (Figure 6.12) are setup appropriately. This is achieved by emulating the VDP's control and data ports. When a port is read or written to by software, the corresponding procedure is called to emulate the properties of the original hardware. Figure 6.13 provides an overview of the operation and interaction of the two ports. Further details of the values written to the ports can be found in section 3.3.1, (p.21).

```
// Video RAM (16K)
private byte [] VRAM;
// Colour RAM (32 Bytes on SMS, 64 on GG)
private byte [] CRAM;
// VDP Registers
private int [] vdpreg;
```

Figure 6.12 – VDP Main Data Structures

```
public void controlWrite(int value)
{
  // First Byte Of Command Word
  if (first_byte)
  {
    first_byte = false;
    command_byte = value; // Store First Byte
  }
  // Second Byte Of Command Word
  else
  {
    first_byte = true;
    // Set VDP Register (use First Byte Value)
    if ((value >> 4) == 0x08)
      vdpreg[(value & 0x0F)] = command_byte;
    // Define Subsequent Read/Write to Control Port
    else
    {
      // Set Whether To Access CRAM/VRAM
      operation = (value >> 6);
      // Set Location in VRAM/CRAM for Read/Writes
      location = command_byte + ((value&0x3F)<<8);
      // Store VRAM in Read Buffer
      if (operation == 0)
      {
        read_buffer = unsigned(VRAM[location++]);
        if (location > 0x3FFF) location = 0;
      }
    }
  }
}
```

```
public void dataWrite(int value)
{
  // Reset Byte of Command Word To Write
  first_byte = true;

  // Switch On Operation Determined By
  // Previous Write to Control Port
  switch (operation)
  {
    // VRAM Write
    case 0x00:
    case 0x01:
    case 0x02:
      VRAM[location] = (byte) value;
      break;
    // CRAM Write
    case 0x03:
      if (master_system_mode)
        CRAM[location & 0x1F] =
          (byte) value;
      else if (game_gear_mode)
        CRAM[location & 0x3F] =
          (byte) value;
      break;
    default:
      break;
  }
  // Increment Location Automatically
  location++;
  if (location > 0x3FFF) location = 0;
}
```

Figure 6.13 – Emulation of VDP Control & Data Ports

Once the VRAM, CRAM and VDP registers are set correctly by the emulated software, the display is ready to be rendered. The display is generated by two main procedures; the first draws the background layer, the second draws the sprite layer. The sprite drawing procedure will be studied, as it is the simpler of the two. The procedure plots all the sprites on the current line (Figure 6.14). The sprite attributes are dependent on the VDP registers, setup during control port writes and the VRAM, setup during data port writes.

```
private void drawSprite(int lineno)
{
① int sat = (vdpreg[5] &~ 0x01 &~ 0x80)<<7;      // Sprite Attribute Table Address
  int count = 0;                                  // Number Of Sprites Drawn On This Line (MAX 8)
  int height = 8;                                 // Height of Sprites (Default: 8 PIXELS)
  boolean zoomed = false;                         // Zoom Sprites (Default: Off)

  if ((vdpreg[1] & 0x02) == 0x02) height = 16;    // Enable 8x16 Sprites
  if ((vdpreg[1] & 0x01) == 0x01)                 // Zoom Sprites
  {
    height <<= 1;
    zoomed = true;
  }
-------------------------------------------------------------------------------------
② for (int spriteno = 0; spriteno < 0x40; spriteno++)// Search Sprite Attribute Table (64 Bytes)
  {
    if (count >= 8)                               // Sprite Overflow (More than 8 Sprites On Line)
    {
      status |= 0x40;                             // Set Sprite Overflow Flag
      return;                                     // and stop drawing this line
    }
    int address = sat+(spriteno<<1);              // Address of Sprite in Sprite Attribute Table
    int x = unsigned(VRAM[address + 0x80]);       // Sprite X Position
    int y = unsigned(VRAM[sat + spriteno]);       // Sprite Y Position
    int i = unsigned(VRAM[address + 0x81]);       // Sprite Tile Index

    if (y == 208) return;                         // VDP stops drawing if y == 208
    y++;                                          // Y is actually plotted at +1 of value
    if ((vdpreg[6] & 0x04) == 0x04) i |= 0x100;   // Use Sprite Tile Index from 100 - 1FFh
    if ((vdpreg[1] & 0x02) == 0x02) i &= ~0x01;   // When using 8x16 sprites LSB has no effect
    if ((vdpreg[0] & 0x08) == 0x08) x -= 8;       // Shift Sprites Left By 8 Pixels
    if (y > 240) y -= 256;                        // If Y is off the screen draw from –16 onwards

    if ((lineno >= y) && ((lineno-y) < height))   // If Sprite Falls On This line
    {
        if (!zoomed)                              // Draw Normal Sprite
        {
-------------------------------------------------------------------------------------
③          int adr = (i<<5) + (lineno-y)*4;       // Address of the 8 Pixels to Plot in VRAM
           for(int bit = 0x80; bit > 0; bit=bit>>1) // Cycle through the 8 Pixels
           {……}                                   // Code to Plot Coloured Pixel
        }
        else {……}                                 // Draw Zoomed Sprite
        count++;                                  // Increment Sprite Count
    }
  } // End For Loop
}
```

Figure 6.14 – Plot One Line of Sprites

① The location of the sprite attribute table is retrieved from bits 1-6 of VDP register 5. (The sprite attribute table stores the location of the co-ordinates at which to plot the sprites, as well as the tile number to use for a particular sprite.)

② This loop searches the 64 bytes of the sprite attribute table. It searches the table from start to end, but plots the sprites in a reversed order (i.e. sprite 0 overlaps sprite 63). The attributes of each sprite are extracted and examined to determine whether the sprite should be plotted. Once 8 sprites have been drawn on the current line, the VDP stops searching through the table, and no more sprites are plotted on the line.

③ Sprites are composed of 8x8 tiles. As the procedure draws a single line, only 8 pixels of a tile are plotted (unless sprite zooming is enabled). The address of the 8 pixels to plot is extracted from VRAM. This is achieved by multiplying the tile index by 32 (the code actually bit shifts the value for speed purposes) and adding the tile row in question to this value. A separate procedure is then called to plot each pixel and extract its colour from CRAM.

The output from the sprite and background rendering procedures is a single array containing a representation of the current frame, encoded using the RGB Java colour model.

The Master System palette (Figure 6.15), is significantly different from the Java palette (Figure 6.16), and colours must be converted between formats. Initially, this was achieved by scaling the Master System palette appropriately as each pixel was plotted to a Java equivalent. The performance of this method was too slow. It was replaced with a pre-calculated lookup table containing the Java equivalent of all possible Master System colours. This produced a 5% increase in the overall speed of JavaGear.

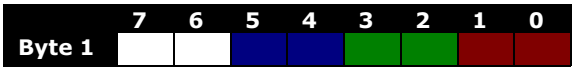


Figure 6.15 -
Sega Master System 8-Bit Encoded Colour

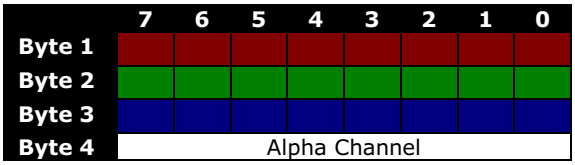


Figure 6.16 -
Java 32-Bit Encoded Colour

The array of pixels, produced off-screen by the Vdp class, is passed to the ScreenPanel class where it is rendered to the screen. This is achieved using Java's *MemoryImageSource* class (Figure 6.17).

```
Image screen =
createImage(new MemoryImageSource(WIDTH, HEIGHT, pixel_array, 0, WIDTH));
```

Figure 6.17 – MemoryImageSource

After the array is updated with pixel data from the latest frame, it is redrawn by the ScreenPanel class (Figure 6.18).

```
// Create Graphics Context for Component
if (g2 == null) g2 = (Graphics2D) getGraphics();
// Fetch Image Data From Source
screen.flush();
// Draw and Scale Image
g2.drawImage(screen, 0, 0, setup.h_end_scaled, setup.v_end_scaled, this);
```

Figure 6.18 – Drawing The Image

The ScreenPanel class overrides the standard methods to paint the component. The component is painted by manually calling the refresh() procedure. This ensures that the screen is only refreshed as necessary, which increased overall performance by 25%.

6.5 – Sound Emulation

Sound emulation, like video emulation, can be divided into three distinct stages. The first stage implements the interface between the Z80 and PSG. The frequency and volume of the four sound channels are set when software writes to the appropriate Z80 port. Under emulation, writing to this port invokes a procedure to program the virtual PSG (Figure 6.19). The Background Research section 3.4, (p.28) details the precise configuration of the value software writes to this procedure.

```
public void write(int value)
{
  if ((value & 0x80) == 0x80)
  {
    switch((value>>4) & 0x07)
    {
        case 0x00: // Set Tone Generator 0 Frequency
          chan0.first_byte = value & 0x0F;
          current_generator = chan0;
          break;
        case 0x01: // Set Tone Generator 0 Volume
          chan0.volume = value & 0x0F;
          current_generator = null;
          break;
        ..... // etc.
    }
  }
  // Set Second Byte of Frequency
  else if (current_generator != null)
    current_generator.frequency =
    current_generator.first_byte | ((value & 0x3F)<<4);
}
```

Figure 6.19 – Procedure To Program The PSG

The second set of procedures emulate the tone and noise generators present in the PSG. The virtual generators output an integer, representing the amplitude of the current sample. Figure 6.20 illustrates the implementation of the square wave generation algorithm first outlined in Figure 4.8. This procedure generates a single sample of the wave (i.e. a value ranging from +15 to -15).

```
public int getSample()
{
  counter -= psg_cycles;

   // Counter has expired
  if (counter < 0)
  {
    // Flip Pos/Neg Amplitude & Reload Counter
    amplitude_flipflop = !amplitude_flipflop;
    while (counter < 0)
        counter += frequency;
  }

  // Invert Output (0x0F is silence on SMS)
  int output_volume = 15 - (volume & 0x0F);
  // Real SMS Doesn't Output Highest Frequencies
  if (frequency > 4)
  {
    // Output +15 to -15
    if (amplitude_flipflop)
        return +output_volume;
    else
        return -output_volume;
  }
  else
    return 0;
}
```

Figure 6.20 – Square Wave Generation

The outputs from the generators are summed, as dictated by the original hardware specification (Figure 6.21).

```
while (samples left to generate)
{
  join = 0;
  join += chan0.getSample();
  join += chan1.getSample();
  join += chan2.getSample();
  join += chan3.getSample();
  // Code to Output Sample
}
```

Figure 6.21 – Sum Channel Outputs

The number of samples to be generated per frame is determined by the following formula:

$$\text{Samples Per Frame} = \frac{\text{Sample Rate (Hz)}}{\text{Frames Per Second}}$$

The final value is output using the Java sound API. The sampled audio package allows sound to be synthetically generated and output. The *SourceDataLine* interface supports streaming data, with a buffer that can be refilled during playback. This is an ideal solution to convert output from the emulated sound processor into real audio (Figure 6.22).

```
// Create SourceDataLine For Audio Playback (Transmits Audio to Speakers)
SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);

// Acquire Necessary System Resources
line.open(audioFormat);

// Output Sound As Soon As There is Data in Buffer
line.start();
```

Figure 6.22 – Creating a SourceDataLine for Streaming Audio

Once the *SourceDataLine* is initialised, the buffer can be filled as necessary (Figure 6.23). Whenever data is placed on the buffer, it is output to the speakers.

```
private void lineWrite(int value)
{
  // Convert the integer to a byte array with one entry
  buffer[0] = (byte) (value);
  // Fill the buffer
  line.write(buffer, 0, 1);
}
```

Figure 6.23 – Fill The Buffer, Output The Sound

A single byte buffer is used to reduce the latency between sound generation and its final output. The offset of a smaller buffer is more intensive CPU usage. This was deemed a necessary trade-off, as the speed reduction was not severe. Tests proved that enabling sound emulation only reduced the overall speed of JavaGear by 8% (Section 10.3, p.66). It can be concluded that the sound emulation is particularly efficient, as it does not burden the emulation with a significant overhead.

6.6 – Component Integration

```
while (running)
{
  // This Loop Generates One Complete Frame
  for (int lineno = 0; lineno < no_of_scanlines; lineno++)
  {
    // Assert Interrupt Line As Necessary For This Line
    vdp.interrupts(lineno);
    // Draw This Line
    if (!frameskip) vdp.drawLine(lineno);
    // Run Z80 For Set Period
    z80.run(cycles_per_line);
  }
  // Output Sound
  sn76496.output();
  // Output Graphics
  if (!frameskip) screenpanel.refresh();

  // Generate Non Maskable Interrupt if pause button pressed
  if (pause_button) z80.nmi();
}
```

Figure 6.24 – Main Emulation Loop

The EmulateLoop class loops the emulation indefinitely and synchronises the components (Figure 6.24). Several optimisations were made to this class to increase overall emulation speed. For example, the Non Maskable Interrupt (NMI) can occur between Z80 instructions, even if interrupts are disabled. Checking for a NMI before every instruction would reduce the speed of JavaGear. The NMI is only used to pause software, thus accurate timing of the instruction is not essential for program execution. The pause button is therefore checked between frames, which maintains performance levels with no visible difference to the emulation.

JavaGear is highly optimised, however, there will be machines that do not posses the processing power to host emulation at full speed. JavaGear includes a workaround to solve this problem, commonly referred to as 'frame skipping'. Frame skipping omits the rendering of a set number of frames per second. For example, if only 20 frames out of the full 60 are rendered per second, JavaGear can run significantly faster. As film is only 24 frames per second, this is still an acceptable rendering rate. Frame skipping is considered a necessary trade-off to enable titles to run at full speed on slower hardware. The code omitted during frame skipping is denoted in Figure 6.24 by the *frameskip* variable.

```
// Time Before Frame Rendered
start_time = System.currentTimeMillis();

// [Generate Frame, Refresh Screen, Output Sound]

// Calculate Actual Length Of Frame
do
{
  actual_frame_length =
        (System.currentTimeMillis() - start_time);
}
// Repeat Until Target Time Reached
while (actual_frame_length < target_frame_length);
```

Figure 6.25 – Restricting Speed

Finally, code was required to lock emulation to PAL or NTSC timing. The time taken to render a frame by the host platform depends on numerous factors, including the speed at which the Z80 instructions are emulated, the Java garbage collector and background operating system activity. Each frame will take an indeterminate time to render on the same platform; there will be vast differences between platforms. To solve speed discrepancies, the time to render a complete frame is recorded and compared against the target time period (Figure 6.25). A loop is called to delay emulation, until the target time has been reached. If the host platform is particularly fast, JavaGear will still run at a constant speed.

7.1 – Application Interface

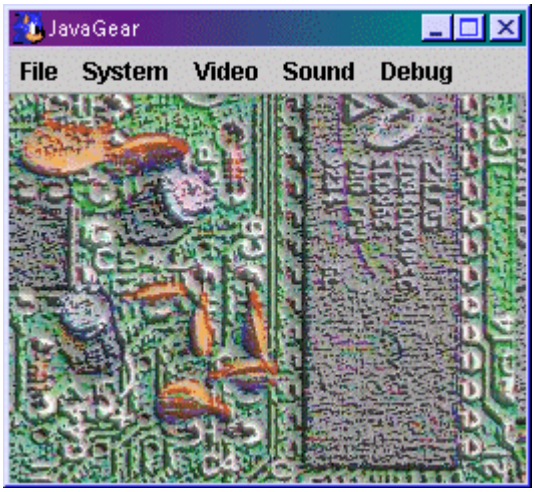


Figure 7.1 – JavaGear Startup Window

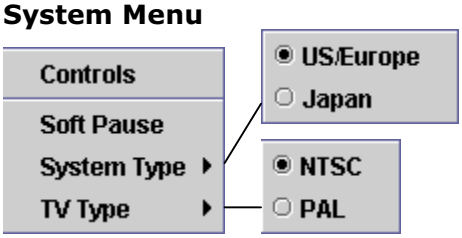
JavaGear's interface was designed to be simple and functional. Its usability is as easy and straightforward as a Master System or Game Gear console. Playing a game involves the single step of selecting a ROM file; the game does not have to be uncompressed. When a game is not loaded, the window contains an image of the Master System's circuit board, creating the illusion of looking into the program's inner workings (Figure 7.1).

JavaGear's menu names are meaningful and logically grouped to aid recognition. The menus are hierarchically structured to ensure the most frequently used operation is at the top of each menu.

File Menu

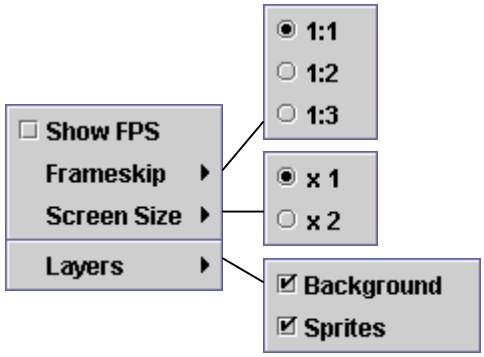
- **Open**
  Loads a Master System or Game Gear ROM from disk.
- **Close**
  Closes ROM
- **Reset**
  Hard Resets ROM
- **Exit**
  Exits JavaGear

System Menu



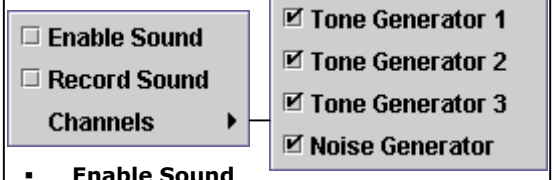
- **Controls**
  Reconfigures mapping between emulated control pad and PC keyboard.
- **Soft Pause**
  Emulates the pause button present on the Master System.
- **System Type**
  Certain software detects the hardware model in use. This setting can cause software to behave in different ways.
- **TV Type**
  Toggles PAL or NTSC timing.

Video Menu



- **Show FPS**
  Displays the number of frames per second rendered and a percentage showing how this compares to the desired speed.
- **Frameskip**
  Omits the rendering of frames to speed up emulation on slower machines.
- **Screen Size**
  Sets the size of the window in comparison to the original Master System display.
- **Layers**
  Toggles emulation of the background and sprite layers.

Sound Menu



- **Enable Sound**

Toggles sound emulation.

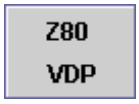
- **Record Sound**

This option records the sound output to a WAV file on the user's hard disk. The option can be enabled regardless of whether sound is enabled.

- **Channels**

Toggles emulation of a particular sound channel.

Debug Menu



- **Z80**

Launches the graphical Z80 debugger. This allows code to be executed instruction by instruction.

- **VDP**

Launches the graphical VDP information frame. This displays information about the current state of the video processor, including register details.

Game Gear titles have a smaller active display area. The portion of the display that is not used contains a scan of a Game Gear console (Figure 7.2).

This has two benefits. Firstly, in applet mode the window cannot be resized, therefore a border of some description would be necessary anyway. Secondly, the user is given a similar viewpoint to playing the original console, enhancing the emulation experience.

Figure 7.2 – Game Gear Mode

7.2 – Controller Configuration

Figure 7.3 –
PC Keyboard to Console Controller Configuration

The interface to map the PC keyboard to the Master System or Game Gear controller is particularly intuitive. A Game Gear graphic is displayed; a key can be mapped to the appropriate button by positioning the cursor over the button and pressing the corresponding key on the PC keyboard (Figure 7.3).

When the cursor is over a button, the button is transparently highlighted, so that the underlying image remains visible. The central Game Gear display area is used to print the current key mapping. The text is highlighted in correspondence with the controller button it is mapped to. The user can revert to the previous configuration at any time by pressing the cancel button.

7.3 – Debugging Interface

JavaGear's Z80 debugging interface allows program code to be executed instruction by instruction, and its effect on the Z80's flags and registers observed. JavaGear's main window is updated appropriately as instructions are executed, clarifying the purpose of portions of the code. It proved an invaluable tool during development in resolving problems with the Z80 emulation. It also provides Z80 programmers with a straightforward method to debug code. The functions provided by the Z80 debugger are presented in Figure 7.4.

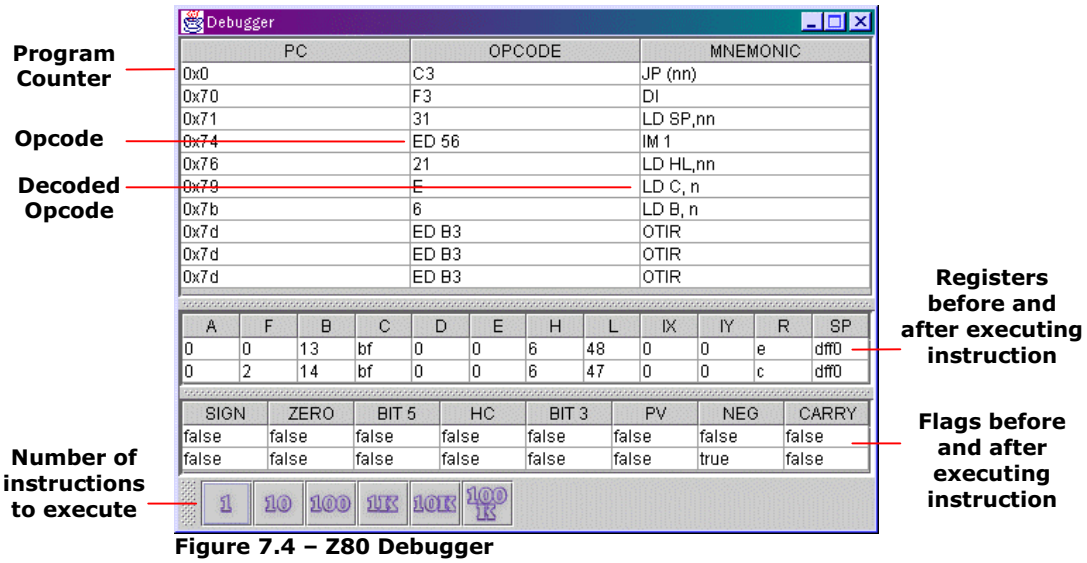


Figure 7.4 – Z80 Debugger

JavaGear's VDP debugger provides information specific to the current video configuration. This includes a full overview of register settings and interrupt details (Figure 7.5).

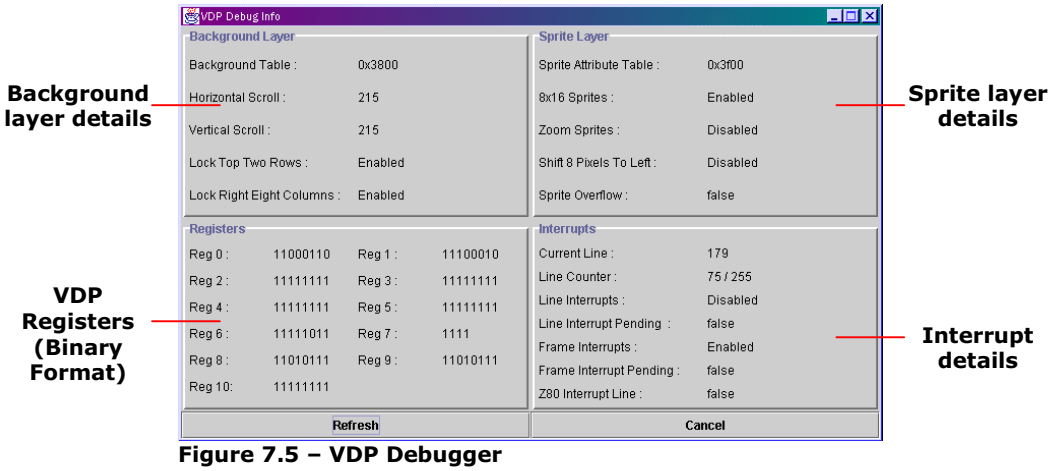


Figure 7.5 – VDP Debugger

7.4 – Applet Interface

JavaGear's applet interface is simpler than the application interface for a number of reasons. It can be assumed that a user running JavaGear through a webpage will not have read the accompanying documentation; it is important that the interface is as self-explanatory as possible. The menu bar was therefore replaced with a toolbar containing a reduced set of features (Figure 7.6). The use of icons rather than English text ensures the interface can be comprehended by different nationalities.

Figure 7.6 – Applet Toolbar

For security reasons, an applet cannot access a user's filespace. In applet mode, certain functionality is not possible, including the recording of sound output to a WAV file. Furthermore, ROM files must be downloaded from the originating server. A replacement ROM selection dialogue was written that listed files stored on the server. As the list of ROM files is configured serverside, it is also possible to specify whether the game requires PAL or NTSC timing when configuring JavaGear for applet use. This eliminates the need for such a setting in the applet interface (Figure 7.7).

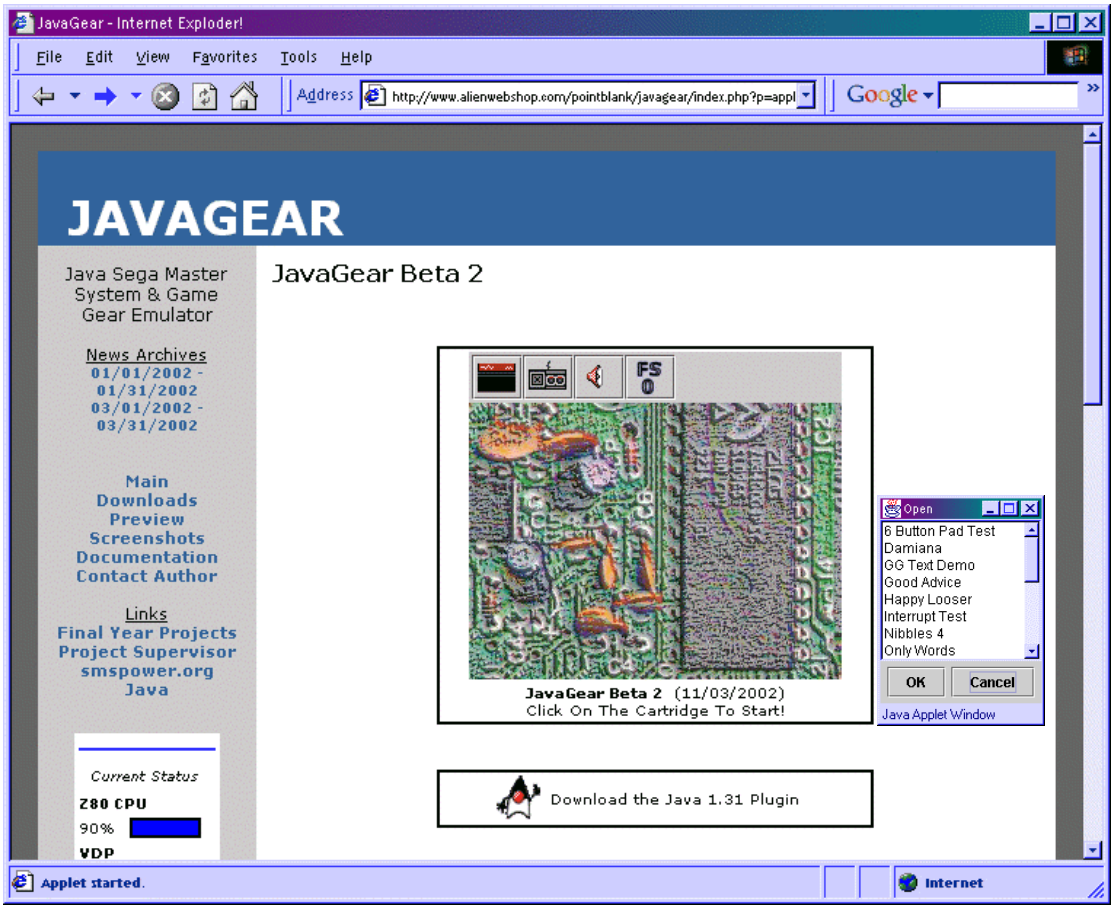


Figure 7.7- Applet Interface

8 PROJECT MANAGEMENT

8.1 – Development Process

JavaGear's management and development was based on the Rational Unified Process. Following the project's inception, the risks involved in the project were analysed to ensure development success.

8.2 – Elaboration

Requirements Risks	Solutions
Sufficient information about key aspects of the Master System's hardware cannot be obtained.	Post queries to a Master System development forum on the Internet. E-mail a Master System programmer, or an emulator author.

Technological Risks	Solutions
A component does not work as expected.	Write Z80 Master System code to test the component. Compare the component's output against that of a working emulator.
A component does not work when integrated in the project.	Write a series of prototypes to verify the component's design and interaction beforehand.
Java is not fast enough to run the emulator at an acceptable speed after optimisation.	Processor speeds will increase. The emulator will still be an interesting theoretical exercise. Consider rewriting the emulator in C or C++.

Skills Risks	Solutions
Documentation on an aspect of the Master System is misinterpreted or not understood.	If the functionality is not essential, consider omitting the feature from the project. If essential, seek clarification from an Internet forum.
The emulator works, but compatibility is poor.	Write debugging methods to output the state of the components.
The emulator is not complete by the project deadline.	The design of the emulator can be discussed, and working components demonstrated with their relevant debugging methods.

8.3 - Construction and Transition

A plan was devised to co-ordinate development progress. The urgent tasks were tackled first to minimise development risk. The initial goal was to produce a working emulator capable of running a range of titles. The order in which tasks were completed deviated from the plan in certain areas. Early work on video emulation proceeded faster than anticipated, although the background and sprite rendering code was not perfected until later.

Task	Urgency	Estimated Completion	Actual Completion
MEM: System Memory Map, with simple ROM loader. (32K ROMs only – No paging)	High	Week 2	Week 1
Z80: Intel 8080 compatible opcodes (250)	High	Week 2	Week 3
Z80: ED prefixed opcodes (70)	High	Week 3	Week 4
Z80: Documented DD prefixed opcodes (70)	High	Week 4	Week 5
Z80: CB prefixed opcodes (255)	High	Week 5	Week 5
MEM: Memory paging registers	High	Week 6	Week 6

VDP: Data and control ports	High	Week 6	Week 1
VDP: Display timing & interrupts	High	Week 7	Week 4
VDP: Background layer	High	Week 7	Week 7
VDP: Sprite layer	High	Week 7	Week 8
VDP: Output graphics to screen	High	Week 8	Week 3
UI: Controller support	High	Week 9	Week 9
Z80: Investigate undocumented opcodes, to see if any games make use of them. If they do, implement them.	Low	Week 10	Week 10

During Christmas, development entered a transitional phase; a public beta was released on the Internet to gather feedback and discover bugs. Development focused on debugging the VDP and Z80 emulation as opposed to implementing new functionality.

Complete pending tasks marked Urgent.	High	Week 1	-
Compile compatibility list	Medium	Week 1	Week 1
Distribute on Internet	Low	Week 1	Week 1
Bug fixes	Medium	Week 3	Week 3

The second term saw the implementation of many new features including sound and Game Gear specific emulation. Sound emulation took longer than planned due to a lack of information concerning the operation of the sound processor. This problem was solved by contacting the programmer of a Master System emulator who kindly explained its undocumented functionality [16].

The 'netplay' feature to enable play against Internet opponents was dropped. It was deemed too ambitious to develop and test in the limited time period available. Optimisation was the final refinement, as it was likely to reduce the clarity of the code, making further development tougher.

SOUND: Emulate SN76489 PSG sound chip	Medium	Week 1	Week 4
Add Game Gear specific emulation	Medium	Week 3	Week 1
UI: Create GUI	High	Week 4	Week 5
NET: Ensure JavaGear runs as applet	Low	Week 4	Week 2
NET: Add ability to play games over Internet	Low	Week 5	-
UI: Create sprite/tile viewer	Low	Week 6	-
Stability/Optimisation	High	Week 8	Week 9

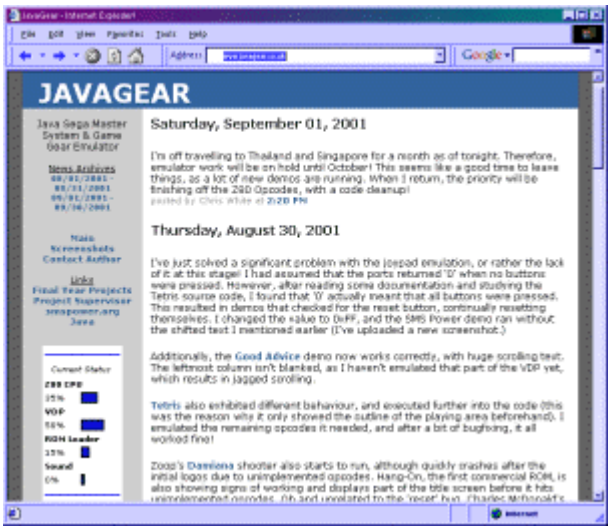


Figure 8.1 – Online Diary (www.javagear.co.uk)

Development decisions and progress were logged in a notebook and an online diary (Figure 8.1). This ensured the reasoning for decisions could be referred back to as necessary.

The JavaGear website presented the opportunity to release public betas of the emulator. The JavaGear betas proved a useful way of regularly producing finished code, suitable for public use. Additional bugs were fixed as a result of feedback from users.

9 RESULTS

The following results aim to display the features developed in this emulation. The screen shots illustrate many of the discussed algorithms in operation. Where appropriate, the equivalent output from the Master System is shown, so that emulation accuracy can be verified. The Master System screen shots were grabbed with a TV capture card, making them appear slightly washed out when compared to the JavaGear screen shots. It should be noted that the outer border has been cropped from the Master System screens.

The *Happy Looser* demo contains an animated 'ripple' effect (Figures 9.1 & 9.2). The effect is generated by setting a different horizontal scroll value between each line. Line interrupts are used to accomplish this. The effect is a useful way to verify interrupts and ensure the line by line rendering engine are functioning correctly.

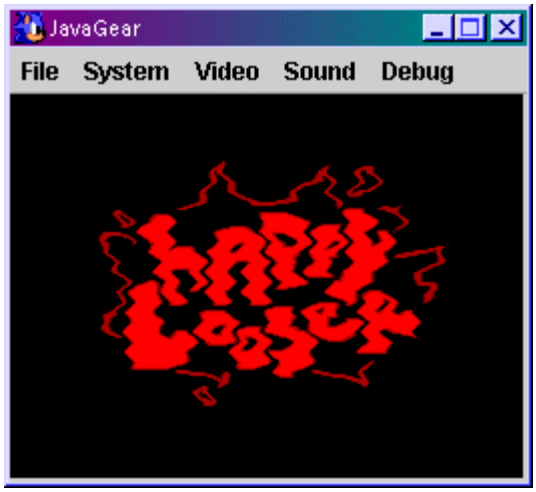


Figure 9.1 – *Happy Looser* on JavaGear

Figure 9.2 – *Happy Looser* on Master System

The following test utility verifies complex interrupt behaviour, where the adjustment of two VDP registers immediately change the state of pending interrupts. If the utility executes correctly, the background turns pink on line 70 (Figure 9.3). However, many emulators incorrectly implement interrupts, and the background turns pink on line 60 (Figure 9.4).

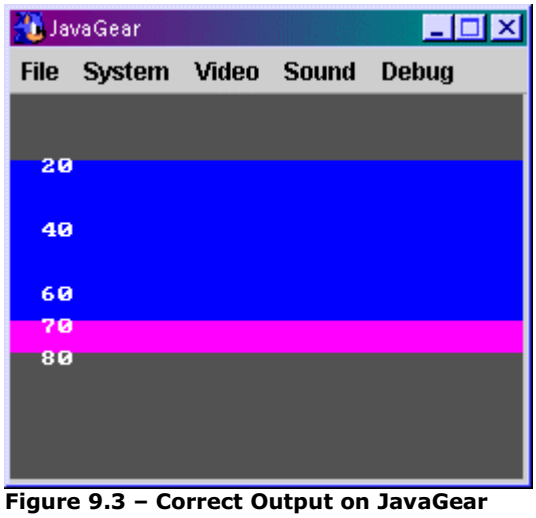


Figure 9.3 – Correct Output on JavaGear

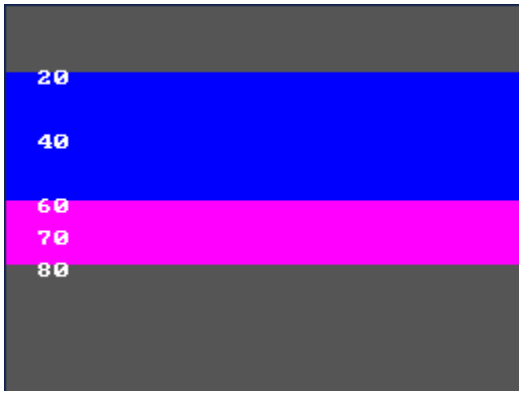


Figure 9.4 – Incorrect Output

Sonic the Hedgehog is one of the most popular Master System titles (Figure 9.6). It was important to ensure it executed accurately on JavaGear (Figure 9.5), as it is likely to be a widely used with the emulator.

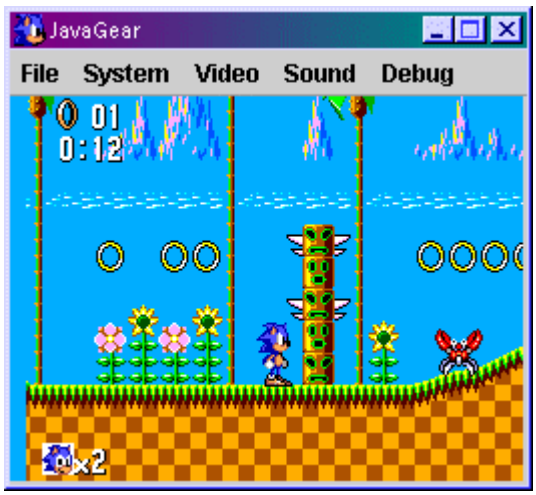


Figure 9.5 –
***Sonic the Hedgehog* on JavaGear**

Figure 9.6 –
Sonic the Hedgehog on Master System

Figure 9.7 –
***Out Run* on JavaGear**

Figure 9.8 –
***Out Run* on Master System**

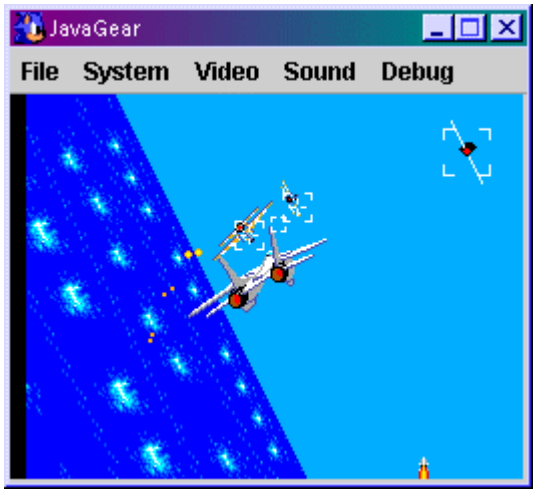


Figure 9.9 –
***After Burner* on JavaGear**

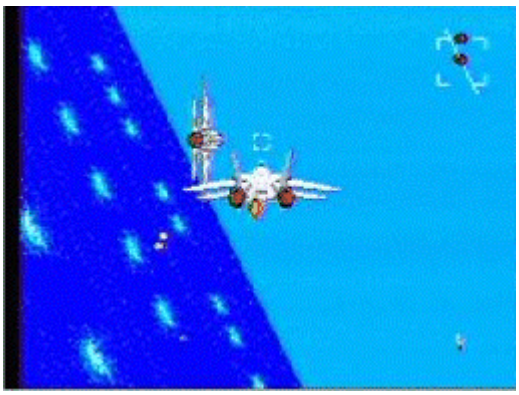


Figure 9.10 –
***After Burner* on Master System**

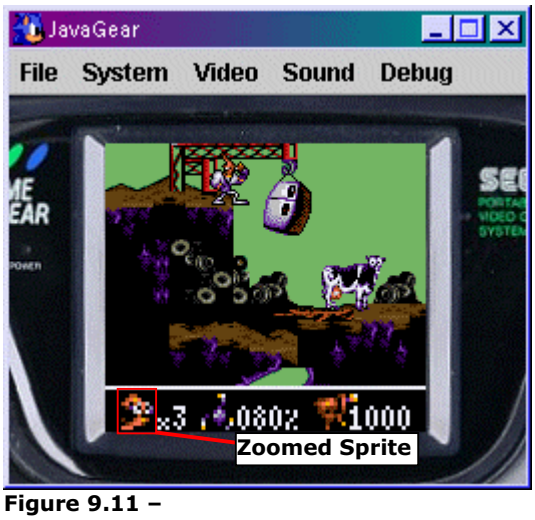


Figure 9.11 –
***Earthworm Jim* With Sprite Zooming**

Some titles use an undocumented VDP feature that doubles the size of sprites (Figure 9.11). Whilst rare, JavaGear implements this functionality to ensure emulation is as accurate as possible.

Figure 9.12 –
Back To The Future 3 With PAL Timing

Some titles require PAL timing to run, including Back To The Future 3 (Figure 9.12). Some emulators fail to run this title as they only emulate an NTSC machine.

10 APPRASIAL

10.1 – Project Success

One of the most formidable tasks facing an emulation author is debugging code to increase compatibility. If a title fails to work as expected, it could be a result of any emulated component from the CPU, to memory mapping or the VDP. A faulty CPU instruction may not be apparent until the software has executed thousands of further instructions. With over one thousand individual CPU operations, simply scanning the code for errors is impossible.

When a title failed to work during the development of JavaGear, the typical process was to log CPU operations to a text file. Logging program execution for a few seconds produces a file of over 50MB in size. Debugging a CPU core is an extremely complex task, and tracking individual bugs can take hours if not days.

JavaGear's high compatibility rate is a testimony to its accuracy, and the continual debugging the project demanded. Hundreds of titles were tested, of which 92% ran successfully. The compatibility of JavaGear is far higher than anticipated considering the short development period, scale and complexity of the project. It is comparable to many of the assembler and C based emulators that possess the advantage of utilising pre-written, highly tuned CPU cores.

JavaGear meets its original specification; it is the first Java based emulator to accurately emulate both the Master System and Game Gear consoles, it boasts an intuitive interface and can be run as either an applet or as an application. It includes functionality beyond that demanded in the specification, including PAL timing, system region selection, sound recording and undocumented VDP functionality.

10.2 – Reliability

The series of public betas ensured the quality of JavaGear reached a high standard. Reports from beta testers indicate that JavaGear runs successfully on a variety of platforms including Windows, Macintosh and Linux.

JavaGear is robust; it does not crash when executing titles which do not function with the emulator correctly. This is achieved with minimal reliance on Java routines to catch exceptions, as doing so reduces performance. Whenever possible, it is ensured procedures are not called with invalid values to begin with. Exceptions are caught in areas where performance is less of an issue, for example in file loading routines. This ensures corrupt or invalid ROM images cannot be loaded into the emulator.

The majority of titles that do not work with JavaGear utilise non-standard memory mapping hardware contained within the game cartridge. The reason certain titles fail to execute is known, and further compatibility improvements could be made to JavaGear by implementing this additional hardware.

10.3 – Performance

Initially, JavaGear rendered 25 frames per second on an 800Mhz x86 based processor. Continual optimisation increased performance to 60 frames per second, matching the original hardware. The frame skipping functionality enabled slower 400Mhz processors to run JavaGear at full speed with sound. JavaGear has also been tested on a 450Mhz PowerPC processor, on which it runs at full speed. JavaGear's performance has ensured that it has become a highly used product, as opposed to a theoretical demo.

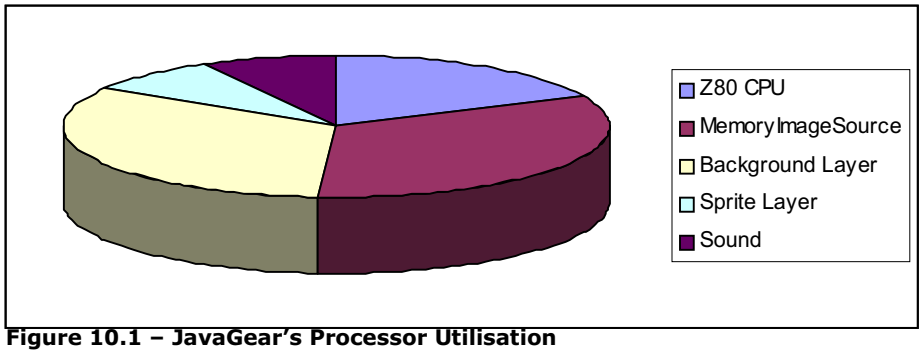


Figure 10.1 – JavaGear's Processor Utilisation

Figure 10.1 provides an overview of JavaGear's processor utilisation. The test was conducted with a graphics demo that continually utilised the hardware scrolling functionality of the VDP. JavaGear can optionally display the speed at which it is operating, ensuring measuring performance is a simple process. It is clear that rendering the background and sprite layers is the most processor intensive emulation operation. However, Java's *MemoryImageSource* class is the greatest performance bottleneck present in the project. Rewriting the video emulation to take advantage of the *BufferedImage* class could significantly enhance performance. This would allow tiles to be cached and the screen layers to be buffered. Unfortunately, some emulation authors have reported that the *BufferedImage* class is unreliable and can result in corrupted graphics under certain circumstances [24].

11 CONCLUSIONS

11.1 – Achievements

JavaGear is the first Sega Master System and Game Gear emulator to be written in Java. It is fast and accurate, fully emulating the core components of the two systems. JavaGear provides functionality not present in many of the existing assembler and C based emulators, including debugging facilities and emulation of unofficial VDP features.

JavaGear is the only Master System and Game Gear emulator to be fully platform independent. JavaGear operates on any platform capable of supporting the Java 1.3 runtime library and above. It is user friendly in terms of operability, providing an intuitive interface combined with the ability to run as a standalone application or as an applet.

The software components comprising JavaGear, (including the emulation of specific processors), have been designed with reusability in mind; they could easily be implemented in other emulation projects. The development of JavaGear will hopefully spur the growth of other Java based emulators in the near future.

11.2 – Deficiencies

Like almost any emulator, there are a small number of titles that do not operate correctly on JavaGear. This could be due to a misinterpretation of documentation or coding errors. Debugging an emulator is a lengthy process; the most successful Master System emulators have been in development for many years. Despite JavaGear's short development period, compatibility is comparatively high and the most popular titles execute correctly.

11.3 – Extensions

Time constraints restricted the scope of the project and certain functionality was omitted. Possible extensions to JavaGear follow:

- JavaGear could be improved by implementing support for the non-standard memory mapping hardware present in certain game cartridges.
- Master System peripherals, including the lightgun and 3D glasses, could be emulated.
- JavaGear provides the potential to include emulation of Sega arcade machines that contain similar hardware to the Master System.
- Java's Internet operability could be utilised to enable games to be played against opponents over the Internet.
- A tile caching engine could be implemented, to increase video rendering performance.

12 REFERENCES

[1] The History of Computing Project
http://64.177.126.115/companies/microsoft/microsoft_1975-1998.htm

[2] Volk, A, M., Stoll, P. A. & Metrovich P.
Recollections of Early Chip Development at Intel
http://www.intel.com/technology/itj/q12001/pdf/art_1.pdf

[3] Nokia: Tools and SDKs
http://www.forum.nokia.com/main/1,6668,030,00.html

[4] MAME – The Multiple Arcade Machine Emulator
http://www.mame.net

[5] The Copyright (Computer Programs) Regulations (1992)
http://www.hmso.gov.uk/si/si1992/Uksi_19923233_en_1.htm

[6] Sega Of America Inc. (1987)
The Sega System Instruction Manual

[7] Cornut, O. (2002)
Meka – Multi Machine Emulator
http://www.smspower.org

[8] MacDonald, C. (2001)
SMS Plus
http://cgfm2.emuviews.com

[9] De Neise, D. (2001)
NESCafe
http://www.davos.co.uk/

[10] Winchurch, D. (2001)
GameBoyEmu
http://www.javaprogramming.co.uk

[11] Millstone, N. (2001)
JavaBoy
http://www.millstone.demon.co.uk/download/javaboy

[12] Mark Knibbs' (1999)
Sega Master System 50/60Hz Modification Pictures
http://home.freeuk.net/markk/Consoles/SMS_mod_pics.html

[13] Zaks, R. (1980)
Programming the Z80
Sybex Inc.

[14] Sega Master System Schematics
http://cgfm2.emuviews.com

[15] The Engineering Staff of Texas Instruments Semiconductor Group
SN76489 AN Data sheet

[16] 'Maxim'
E-Mail: sexymaxim@hotmail.com
Subject: Noise Generation
Date: 24th January 2002

[17] The Space Harrier Absymbel
http://freeweb.pdq.net/brandonmarks2/spaceharrier/gallery.html

[18] Gordon, M. (mike@mikeg2.freeserve.co.uk)
SMSReader
http://smspower.org/smsreader/

[19] Sharp, D. (2001)
Tarmac – A dynamically recompiling ARM Emulator (2001)
University of Warwick
http://www.dynarec.com/~dave/tarmac/

[20] Gordon, M. (mike@mikeg2.freeserve.co.uk)
Only Words Demo
http://www.mikeg2.freeserve.co.uk/

[21] Fowler, M. & Scott, K. (2000)
UML Distilled (Second Edition)
Addison Wesley Longman Inc.

[22] Gosling, J., Joy, B., Steele, G. & Bracha G. (2000)
The Java Language Specification (Second Edition)
Sun Microsystems, Inc.

[23] goldGB Development Archive #3
http://www.goldroad.co.uk/

[24] Duijs, E.
E-Mail: erikduijs@yahoo.com
Subject: Re: MemoryImageSource etc.
Date: 13th March 2002

13 BIBLIOGRAPHY

Bogumil, J. (1998)
Sega Master System FAQ Version 2
http://www.gamefaqs.com

Cool Edit Pro 1.2 Glossary (1999)
Syntrillium Software Corporation
http://www.syntrillium.com

Dix, A., Finlay J., Abowd G. & Beale R. (1998)
Human-Computer Interaction (Second Edition)
Pearson Education Limited

Ess, A. (1999)
Icarus Productions – Z80 FAQ
http://icarus.ticalc.org/articles/z80_faq.html

Fayzullin, M.
HOWTO: Writing a Computer Emulator
http://www.komkon.org/fms/EMUL8/HOWTO.html

Hearn D. & Baker M. (1997)
Computer Graphics C Version (Second Edition)
Prentice Hall International, Inc.

Holub, A. (2000)
UML Reference Card
http://www.holub.com

Horstmann C. & Cornell G. (1999)
Core Java 2 Volume 1 – Fundamentals
Sun Microsystems Press

Horstmann C. & Cornell G. (2000)
Core Java 2 Volume 2 – Advanced Features
Sun Microsystems Press

Kernighan, B. W. & Ritchie D. M. (1988)
The C Programming Language (Second Edition)
Prentice Hall Software Series

Leventhal, L. A. (1979)
Z80 Assembly Language Programming
Osborne McGraw-Hill

MacDonald, C. (2000)
SMS/GG VDP documentation (Version 4.5)
http://www.cgfm2.emuviews.com

MacDonald, C. (2001)
SMS/GG Hardware Notes
http://www.cgfm2.emuviews.com

MacDonald, C. (2001)
Sega Master System VDP documentation
http://www.cgfm2.emuviews.com

Pettus, S. (1999)
The EmuFAQ
http://www.emuhq.com/emufaq/

Pettus, S. (2000)
SegaBase – Sega Master System & Game Gear
http://www.atani-software.net/segabase/SegaBase-MasterSystem.html

Ponder, J. (1998)
Generator – A Sega Genesis Emulator
University of Warwick

S8-Dev Forum
http://smspower.org/dev/forum/

Scherrer, T. (1999)
Z80 CPU Hardware and Software FAQ
http://www.geocities.com/SiliconValley/Peaks/3938/

Silberschatz, A., Galvin & P. Gagne G. (2000)
Applied Operating System Concepts (First Edition)
John Wiley & Sons Inc.

Surdulescu, R. (2001)
Emulation in Java
http://home.austin.rr.com/razvans/projects/jzx/article/jzx.htm

Talbot-Watkins, R. (1998)
Sega Master System Technical Information
http://smspower.org/dev/docs/

Winchurch, D. (2001)
GameBoyEmu Final Year Report
University of Birmingham

Young S. (2001)
The Undocumented Z80 Documented (Version 0.4)
http://www.msxnet.org/tech/

14 APPENDIX I

14.1 – How To Run JavaGear In Application Mode

JavaGear is obtainable from two locations:

- JavaGear can be downloaded from: **www.javagear.co.uk**
- JavaGear can be found in the following directory on the university network: **ug85cjw/javagear/**

JavaGear is run as follows:

- **java –jar JavaGear.jar**

Performance can be increased by invoking Java's incremental garbage collector, which is less likely to delay screen updates:

- **java –jar –Xincgc JavaGear.jar**

A Master System or Game Gear binary is loaded by choosing the Open option from the File menu. A selection of public domain demos have been placed in the following directory: **ug85cjw/javagear/roms/**

Further operational details can be obtained through referral to the User Interface section of this report (Section 7, p.56).

14.2 – How To Run JavaGear In Applet Mode

JavaGear can be run as an applet by opening the following webpage: **www.javagear.co.uk**

Follow the **Preview** link in the leftmost column and the applet will load automatically.

The server has a selection of public domain demos that can be run by selecting the cartridge icon from the toolbar.

14.3 – Source Code

The source code is located in the following directory on the university network: **ug85cjw/javagear/source/**

The source code can be automatically documented with the javadoc utility:

- **javadoc -d .\docs *.java**

The documentation will be generated in the **docs** subdirectory.

15.1 – External Project Success

The public Internet releases of JavaGear have proved a phenomenal success. The JavaGear website (www.javagear.co.uk) has been visited over 10,000 times.

News about JavaGear has appeared on emulation web sites worldwide, (Figure 15.1).

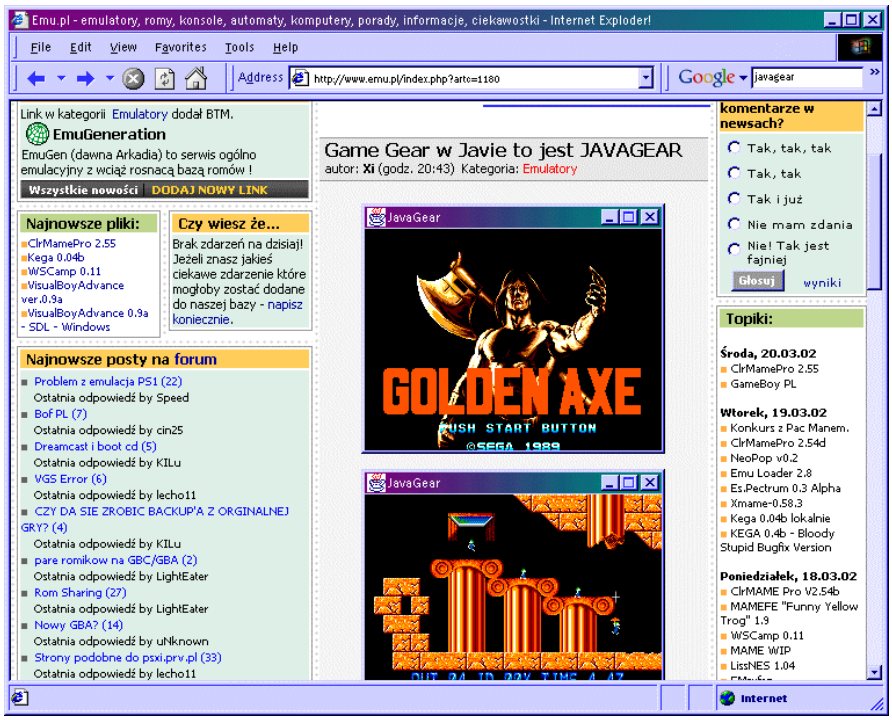


Figure 15.1 –
An Article On A Polish Web Site About JavaGear (www.emu.pl)

JavaGear was described on the S8-Dev Master System development forum (smspower.org/dev) as "awesome, compatibility wise".

It is expected that development on JavaGear will be continued in the near future.