The First ATARI ST Book

A Data Becker
Book from
First Publishing
Limited

1st
FIRST PUBLISHING LTD

PRESENTING
the
ATARI ST

By L. English and J. Walkowiak

A DATA BECKER BOOK

Published by

1st

FIRST PUBLISHING LTD

All instructions, technical instructions and programs contained in this book have been carefully worked out by the authors, i.e., written and run under careful supervision. Every effort has been made to include complete and accurate information. However, First Publishing assumes no responsibility for its use. The authors will always appreciate receiving notice of subsequent mistakes.

Printed in U.K. Translated by Greg Dykma
Edited by Jim D'Haem
Drawings by Russ Taber

ISBN 0948015-950

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Preface

A new era is beginning in the world of computers. The line which separates the home from the business computer is becoming less distinct as machines such as the Atari ST appear. These new super-machines carry a price tag aimed at the home user - especially those who want to upgrade from their present 8-bit home computer. But because of the high performance capabilities these new computers, they can also be used for serious business, scientific and engineering applications.

Although the Atari ST comes to the market at a price which is extremely attractive, its specifications and capabilities set totally new standards.

* The 16/32 bit Motorola 68000 microprocessor offers very powerful processing capabilities found only in considerably more expensive business and scientific computers.

* The 512K of RAM is very adequate for most users and applications.

* GEM, the new graphic oriented user interface, enables novices to quickly become productive with the computer.

The ingredients of Atari's ST are already found in other computers. By themselves these features are not so spectacular, the real innovation, however, lies in the low price at which the whole package is offered.

Ford's Model T was no great technical innovation. But the efficient production techniques and the marketing strategy made the Model T very affordable to the consumer. That made the Model T so successful for Ford. So says Atari of their ST.

Innovation here, represents a leap in the price to performance ratio of home computers. This kind of innovation is a tradition for one of the most significant "movers" in the computer world, Atari Chairman Jack Tramiel.

His first success in home computers was the PET 2001, one of the first out of the box and ready-to-run machines. It's popularity proved so great that enthusiasts had to wait months to buy one, after it was introduced in 1977. But the PET's success was soon overshadowed by that of the VIC-20. The VIC-20 was a smashing success owing to its fine color graphics, sound synthesis capabilities, available peripherals and most of all its affordable price. Millions were sold. To prove that he was not spoiled by success, Tramiel revealed his second masterpiece, the Commodore 64. The '64 remains the most successful home computer to date. More than four million owners are convinced that at such an attractive price, the '64 is ***the*** computer that meets their needs.

Now Atari is hoping that the ST will be Tramiel's third masterpiece. At the original price of the Commodore 64, the ST offers very exciting performance features of much more expensive computers. To the general public, the ST offers a new spectrum of very ambitious computer applications that were previously unavailable because of price considerations. Home computer users can tap completely new areas with the ST, while business users may quickly find that they can replace considerably more expensive computers with the Atari ST and thereby save a good deal of money too!

The goal of this book on the Atari ST is not only to provide the reader with a summary of the capabilities and features of this fascinating new machine. It should also serve as a good source information for prospective buyers. Additionally, we hope that it will suggest potential uses for this powerful new computer.

The information in this book is book is derived from hands-on experience which the authors received during their work with prototypes of the new Atari ST. As you are aware, computer development is a creative and on-going process. For this reason, the production models and the accompanying software may deviate slightly from the information presented in this book.

We wish to thank the Atari Corporation for making the ST available to the authors and for their technical assistance. Good luck, Atari!

1. INTRODUCING THE 16-BIT PROCESSORS

Eight-bit microprocessors appeared on the market in the middle of the 70's. It wasn't long before these processors became the basis for the first home computers. Among the first were the PET 2001 (from Commodore), the Apple (from Apple Computer) both of which used the 6502 processors, and the TRS-80 (from Tandy) which used the Z-80 processor. These micros had from 4K to 16K of memory, a monochrome screen that displayed 25x40 or 16x64 characters and a built-in BASIC interpreter. They were priced at about one thousand dollars.

Now, for less than one-fourth of the price, you can get a computer with 64K of memory, built-in BASIC, high-resolution color graphics and excellent sound capabilities. This highlights the incredible improvement in performance and price that has taken place in such a short time. These were made possible mainly through advances in semiconductor technology.

For example, in the late 1970's increasing the memory capacity of an 8K PET to 32K cost well over £300. Today, 64K of RAM costs less than £25. The prices of peripherals have also fallen dramatically. In 1980, a printer could hardly be bought for less than £750. Today, for about £250, you can select from a wide variety of printers with far superior features and performance than was available only a few years earlier. This also holds true for mass storage devices. A dual disk drive with a 340K capacity cost £1200 three years ago. Today, for about £600, you can buy 2 megabytes of mass storage.

Why does it seem that we always need more and more storage capacity? In the past, mainframe computers typically had main memory capacities of from one-half to several megabytes. Using these computers, it was possible to execute computation intensive mathematical programs, usually written in the FORTRAN programming language, or to perform data intensive commercial tasks, most often written in the COBOL programming language. The compiler alone often required as much as 100K to run in such installations. If in 1975 you asked if FORTRAN programs could be run on a home computer, the answer was definitely: "NO". But by 1980, FORTRAN and COBOL compilers were available for some microcomputers. The capabilities of these compilers was somewhat limited, but is was possible to use the huge base of existing FORTRAN and COBOL programs (these programming languages have been around since the fifties) on a microcomputer. But in so doing, the limitations of micros also became clear.

The execution speed of compiled programs is considerably greater than interpreted languages such as BASIC. Therefore complex problem solutions which require a few minutes on a mainframe often take several hours on a micro. Even when speed of execution is not a concern, the limited amount of memory - 64K - often prohibits the use of micros for some applications. To understand why 64K bytes represents the limiting boundary for most 8-bit microprocessors, we'll take a brief excursion into the hardware end of microprocessors.

When a microprocessor wants to access memory, it must determine which data it wants to read. To do this, each location in memory is referenced by a number or *address*. The processor has 16 address lines, each of which can assume one of two different conditions. Therefore you can select two different memory locations with each address line. If there are two lines available, then there are four different combinations for selecting or addressing the memory. With the sixteen address lines available, $2^{16} = 65536$ different memory locations can be addressed. Since $2^{10} = 1024$ addresses represent one kilobyte (K), 65536 is 64K. Since an 8-bit processor has eight data lines, each address can contain one of 256 (2^8) different values. A collection of eight bits is called a byte.

To go beyond these boundaries with an 8-bit processor requires some tricks. If you need more than 64K of memory for a program and/or data, then a section of memory not being used at that moment can be saved to a mass storage device, such as a disk, and reloaded later when required. The disadvantage of this procedure is that it is considerably slower than accessing the data directly in memory. The processor can access a memory location in mere microseconds (millionths of a second) while disk drive access (to retrieve the saved data) requires much more time and slows the execution of the program.

Another method of accessing more than 64K or memory is by *bank-switching*. Here, several memory banks each containing 64K are used. Using a software-controlled switch, you can select between the different banks and thus gain access to more than 64K. This technique is certainly faster than the previous method, but complicates data access since the software must be specially written to switch between the memory banks.

For these reasons, people soon started looking for ways to improve the efficiency and capability of microprocessors. There were two major concerns: greater processing speed and the access of more memory.

A first step in the direction of greater memory access resulted in an improvement of the bank-switching technique. Switching between different memory banks was assigned to the processor instead of special software routines. The improvement uses a *segment register.* For example the 8086 and 8088 processors contain four segment registers. Using these registers, a 64K segment can be moved in main memory, which can amount to One million bytes (1M). Following each address, a segment register specifies the actual memory access: the code segment register determines from which segment the program instructions are to be fetched; the data segment register specifies the segment where the data is to be found. The stack segment register specifies the location of the stack and an extra segment register can point to an additional 64K segment. The disadvantage of this method is that only 64K can be accessed in one pass. The appropriate segment register must be reloaded in order to access memory outside of this range.

To increase the execution speed of the processor, the internal registers were widened from eight to sixteen bits. This allows operations, such as addition and subtraction, in the processor to be carried out on sixteen bits at a time. To retain this advantage during data transmission, the data bus was also widened from eight to sixteen bits in the case of the 8086. The 8088 is the low-cost version of the 8086. Accordingly, the data bus was retained at a width of eight bits. This permits a smaller size chip and reduces the number of other components, such as bus drivers, required on the board. The penalty is the reduced throughput, since sixteen-bit data must now be transferred with two sequential eight-bit transfers. Viewed in this manner, the 8088 no longer offers much advantages over an eight-bit processor, apart from an improved instruction set. This is why a BASIC program on the IBM PC runs only slightly faster than on many CP/M computers which have an 8-bit Z-80 processor.

1.1. THE TRUE 16-BIT PROCESSORS

Based on these considerations, the engineers at Motorola started the development of a sixteen-bit processor which would overcome these limitations. The goals were: a larger address range, greater data throughput, and a powerful instruction set which is both easy to learn and easy to use. The result of this development was the 68000, samples of which were already available as early as 1979. The 68000 is considered by many as the most capable of the sixteen-bit processors. It has already become the standard in many industrial applications. Because it forms the heart of the Atari ST, we'll examine the 68000 in greater detail in the next section.

Fig. 1-1: 68000 BLOCK DIAGRAM

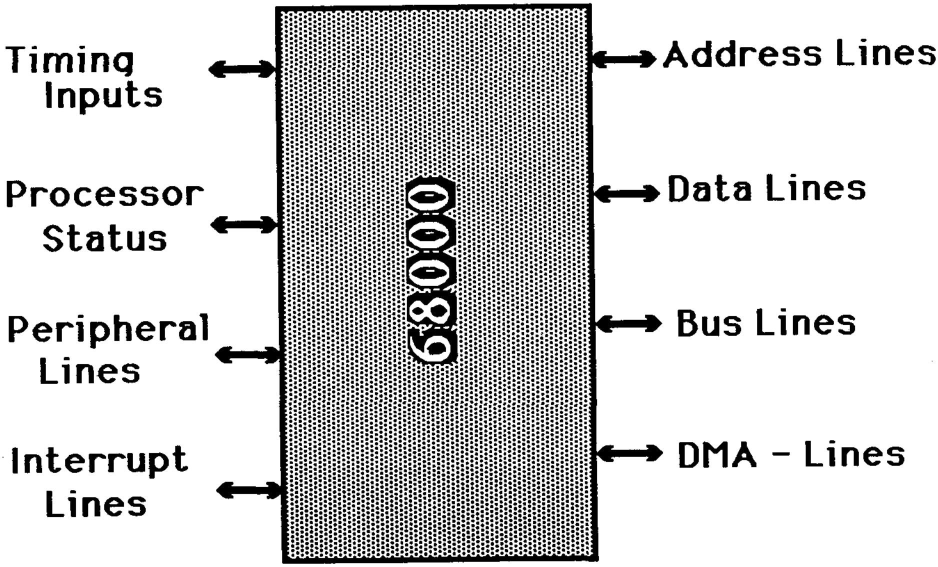
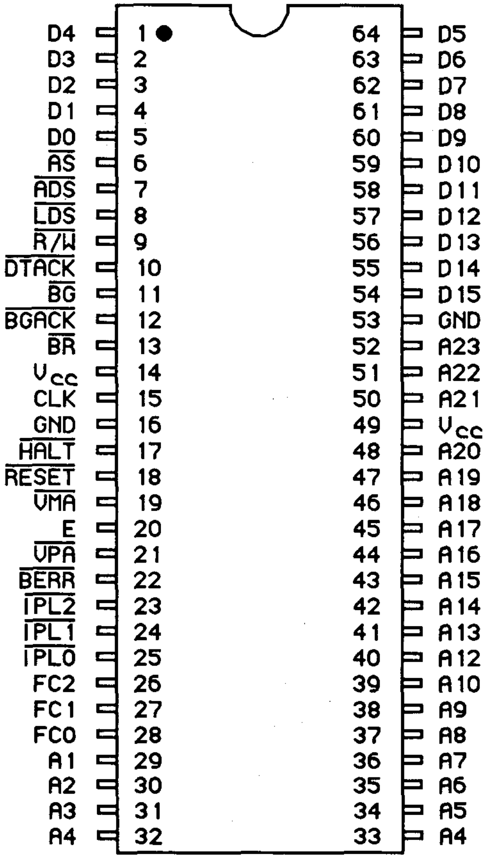


Fig. 1-2: 68000 PIN LAYOUT


D4 1 ● 64 D5
D3 2 63 D6
D2 3 62 D7
D1 4 61 D8
D0 5 60 D9
$\overline{AS}$ 6 59 D10
$\overline{ADS}$ 7 58 D11
$\overline{LDS}$ 8 57 D12
$\overline{R/W}$ 9 56 D13
$\overline{DTACK}$ 10 55 D14
$\overline{BG}$ 11 54 D15
$\overline{BGACK}$ 12 53 GND
$\overline{BR}$ 13 52 A23
V_{cc} 14 51 A22
CLK 15 50 A21
GND 16 49 V_{cc}
$\overline{HALT}$ 17 48 A20
$\overline{RESET}$ 18 47 A19
$\overline{VMA}$ 19 46 A18
E 20 45 A17
$\overline{VPA}$ 21 44 A16
$\overline{BERR}$ 22 43 A15
$\overline{IPL2}$ 23 42 A14
$\overline{IPL1}$ 24 41 A13
$\overline{IPL0}$ 25 40 A12
FC2 26 39 A10
FC1 27 38 A9
FC0 28 37 A8
A1 29 36 A7
A2 30 35 A6
A3 31 34 A5
A4 32 33 A4

1.2. THE 68000 PROCESSOR

In this section, we'll look closely at the makeup and operation of the 68000 processor. This will give you a better idea of the capability of the Atari ST. We'll also compare its operation to conventional eight-bit processors.

If you take a look at the circuit board in the Atari ST, you cannot overlook the 68000. It is the largest integrated circuit and is housed in a 64-pin dual-inline package. Why does the 68000 need so many "legs" (pins)? As previously mentioned, in order to transmit data at high speeds, the data must be sent in parallel, not in two sequential portions. Since the 68000 is a 16-bit processor, we need sixteen data lines. In order to address a large address range, the 68000 requires 24 address lines. This allows 2^{24} memory locations to be addressed. This is equivalent to 16,777,216 or 16 Mbytes, which is 256 times as much memory as a normal eight-bit processor can address. This huge address space is adequate for most any applications implemented on a microprocessor. As a comparison, a huge mainframe computer of the IBM System/370 can directly address 16 Mbytes.

So we find that 40 pins are needed just for the address and data lines. The remaining 24 pins are used for bus control, for the DMA access, for interrupt control, to output the processor status, and for peripheral control. A power source (only +5V) and the clock input are also required. The diagram in Figure 1-1 represents this graphically.

The effectiveness with which the processor can be programmed depends on the capability of the instruction set and on the number and type of internal *registers* available to the programmer. A register is a memory location within the processor which can be accessed at extremely fast speeds and in which most of the operations which the processor can perform are carried out. This explains why the efficiency of a processor increases with the number and size of its registers. To compare the 68000 processor to conventional eight-bit processors, Figure 1-3 shows the register set of the 6502 while Figure 1-4 shows the register set of the 68000. As you can see the 68000 has many more registers than the 6502. Additionally all of the 68000's registers are wider than 8 bits and can therefore process more data per operation.

Fig. 1-3: 6502 REGISTER SET

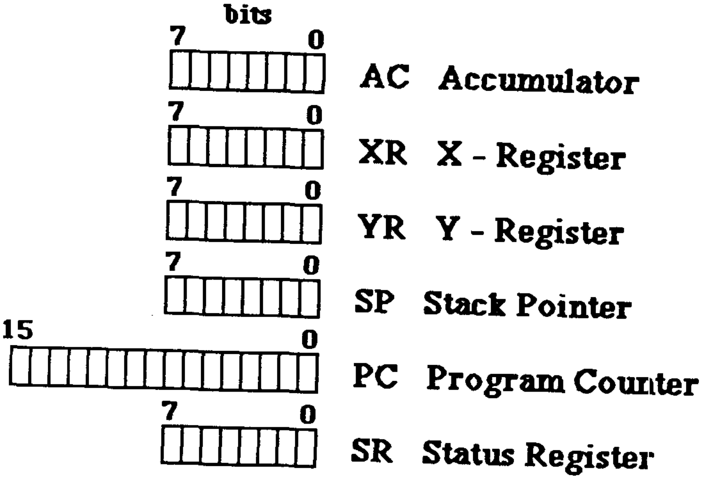
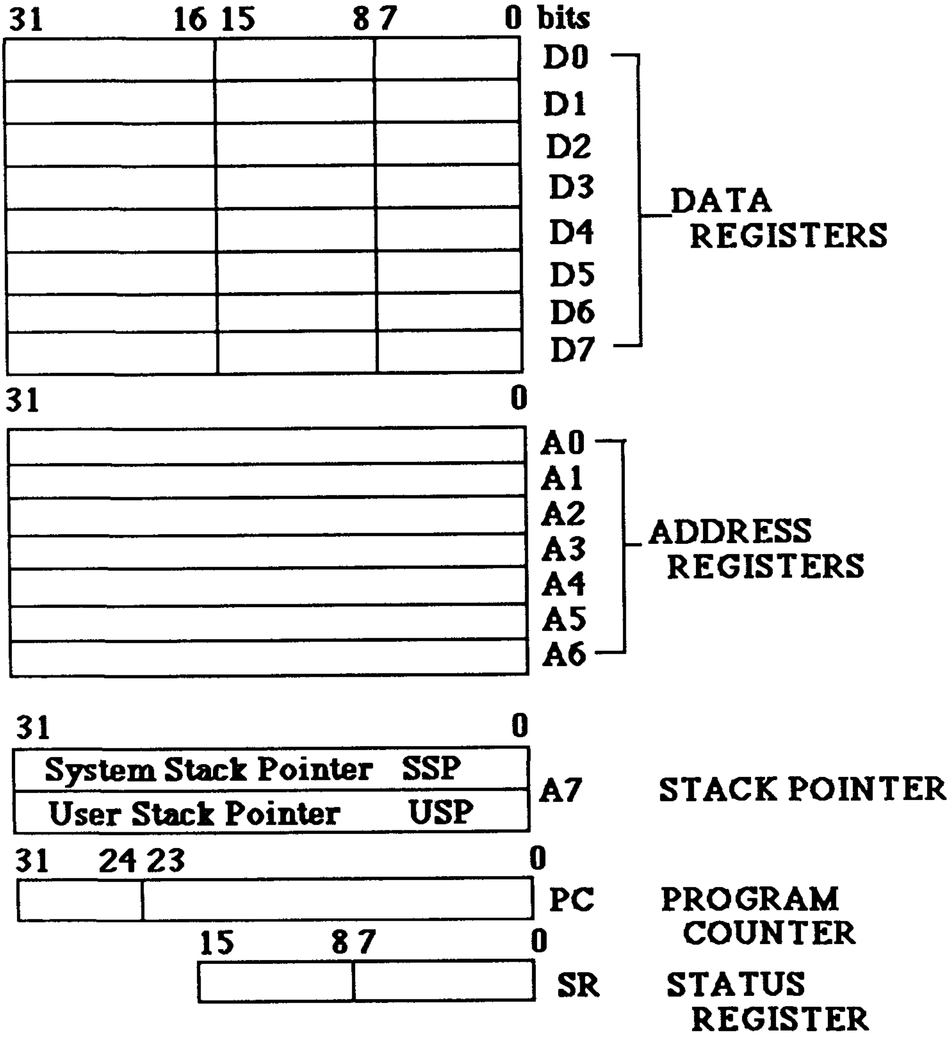


Fig. 1-4: 68000 REGISTER SET



As you can see, all of the registers in the 68000 are 32 bits wide. The register set is divided into eight data registers and eight address registers. Register A7 is available for two tasks and has a special significance. More about this later. The program counter is also 32 bits wide, although only the lowest 24 bits can be used. The program counter (PC) always points to the next instruction to be executed. The status register is sixteen bits wide and is divided into a user byte and a system byte.

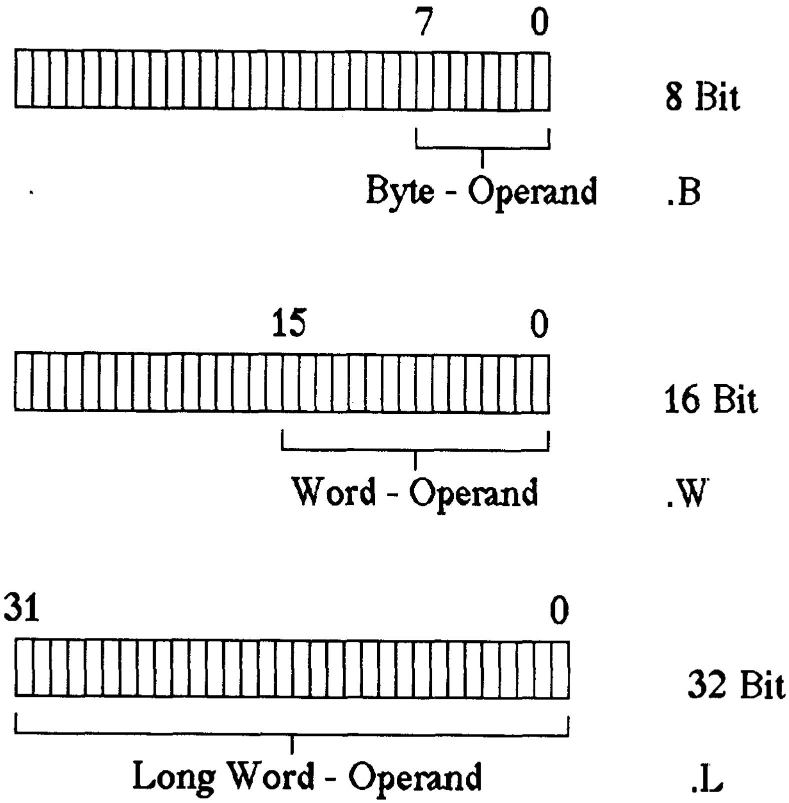
As shown in the Figure 1-5 the data registers are all 32 bits wide too. But they can also be used as either sixteen or eight bit registers. If the contents of two data registers are to be added, for instance, you can specify whether the instruction will operate with eight, sixteen, or 32 bits.

When performing eight-bit processing, only the lowest eight bits are used; bits 8 through 31 remain unchanged and are not affected by the operation. If you use the 16-bit or *word* version of the instruction, only the lowest 16-bits are included in the operation and the upper 16 bits (16-31) are not affected. The 32-bit or *long-word* instruction uses the entire register.

These examples affect only two registers. To transfer data from a register to memory or from memory to a register, you can also specify the processing width. Since the processor has 16 data lines, transferring a single word (16- bits) is the easiest to accomplish. If only one byte (8- bits) are to be transferred, only data lines D0 to D7 are used. The processor informs the rest of the hardware via the bus control signals that only 8 bits will be transferred and that the contents of data lines D8-D15 is irrelevant. This means that it takes the same amount of time to transfer one byte of data as it does to transfer one *word* of data. But to transfer 32-bit data, it is no longer possible to transfer a single *long-word* in one move. First the processor transfers the most-significant word (bits 16-31); the address is automatically incremented; and then the least-significant word (bits 0-15) is transferred. The processor does this all automatically. To the user it looks as if he is working with a 32-bit processor. In this sense, the 68000 can be called a 32-bit processor by the same justification that the 8088 is said to be a 16-bit microprocessor.

The long-word transfer takes more time than the transfer of a 16-bit word, but it is faster than if you had to program the transfer of two words separately. Figure 1-5 clarifies the transfer methods.

Fig. 1-5: 68000 OPERANDS



1.3. THE INSTRUCTION SET OF THE 68000

To make full use of the 68000 hardware features, we need a capable instruction set. And to insure widespread use, this instruction set should be as easy as possible to learn.

These goals are achieved firstly by having a relatively small number of basic instructions. Secondly, most of these have slight variations to work with byte, word, or long-word data. So the programmer needs to learn only a few instruction mnemonics. These mnemonics, may be appended by ."B" for byte, ".W" for word, or ".L" for long-word. The result is a simple, flexible instruction set. The operation codes for these instructions are each sixteen bits long. While this theoretically allows for up to 65536 (2^{16}) different instructions, this is clearly unpractical. The following table lists the mnemonics for the 68000 instructions.

Table 1-1: THE 68000 INSTRUCTION SET:

ABCD	Add Decimal with Extend
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
Bcc	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
DBcc	Decrement and Branch Conditionally
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive Or
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack (reserve stack area)
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move Source to Destination
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal With Extend
NEG	Negate
NOP	No Operation
NOT	One's Complement
OR	Logical Or
PEA	Push Effective Address
RESET	Reset External Devices

ROL	Rotate Left without Extend
ROR	Rotate Right without Extend
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
Scc	Set Conditional
STOP	Stop processor
SUB	Subtract
SWAP	Swap Data Register halves
TAS	Test and Set Operand
TRAP	Trap
TST	Test Byte
UNLK	Unlink (free stack area)

The 68000 recognizes the following conditions ("cc" in the instruction set):

The conditions marked with a * apply to two's complement arithmetic.

EQ	equal
NE	not equal
MI	negative
PL	positive
GT*	greater than
LT*	less than
GE*	greater than or equal
LE*	less than or equal
HI	higher than
LS	lower than or equal
CS	carry set
CC	carry clear
VS*	overflow
VC*	no overflow
T	always true
F	always false

So the 68000 instruction set contains no more mnemonics than a processor such as the 6502. This simplifies learning its machine language. The special capability of the 68000 lies in the numerous addressing methods as well as the power of the individual instructions. As you can see from the table, the 68000 has instructions for multiplication and division. These instructions are found in very few 8-bit processors. In order to briefly show you the different addressing modes, we'll illustrate them using what is probably the most universal instruction, MOVE.

The 68000 generally operates as a two-address machine. This means that a source (where the data comes from) and destination (where the data or result is placed) must be specified for each instruction.

The following instructions use the syntax of the 68000 assembler and demonstrate the different addressing modes.

```
002000  3200            MOVE    D0,D1
```

This instruction copies the contents of register D0 to register D1. This instruction and those that follow, operate on a 16-bit word.

```
002002  323C0007        MOVE    #$0007,D1
```

This instruction loads the constant hexadecimal $0007 into data register D1. As you see, this instruction is two words long. The opcode is $323C,followed by the constant $0007.

```
002006  7207            MOVEQ   #7,D1
```

This instruction has the same effect as the previous one. The "Q" appended to the mnemonic stands for "quick." As you see, this instruction fits in only one word. It is limited to one-byte constants. The operand (07) is built into the opcode and makes the instruction shorter and faster to execute.

```
002008 3208             MOVE    A0,D1
```

In this example, the contents of address register A0 are copied to data register D1. Data registers and address registers can be treated similarly.

```
00200A  32381000        MOVE    $1000,D1
```

In this example, the contents of memory location $1000 are loaded into data register D1. Since the address can be represented as a 16-bit value, we speak of short addressing, similar to zero-page addressing on the 6502 processor.

```
00200E  31C11000        MOVE    D1,$1000
```

In this example, the contents of data register D1 are placed at address $1000. This is, in contrast to the previous example, and transfers data from the processor to memory.

```
002012  3080            MOVE    D0,(A0)
```

In this example, we are using indirect addressing. The contents of register D0 is stored at the memory location whose address is contained in address register A0. This again is a one-word instruction.

```
002014  31400032        MOVE    D0,50(A0)
```

In this example, the addressing mode is similar to the previous one. The address at which the contents of D0 are stored is the sum of the contents of the address contained in register A0 plus the constant 50. You may think of A0 as containing the starting address of a table in which the 50th items is accessed.

```
002018  31801032        MOVE    D0,50(A0,D1)
```

In this example, the addressing mode is further modified. The destination here is the sum of the contents of A0 and D1 and the constant 50.

```
00201C  33C000100000 MOVE    D0,$00100000
```

In this example, we return to the absolute addressing mode. This time the address cannot be represented as a 16-bit word, so an additional word is needed. This addressing mode is called absolute long.

```
002022  303A000A        MOVE    10(PC),D0
```

Here, the program counter's relative addressing permits you to write relocatable programs. The address of the code to be executed is calculated from the current contents of the program counter plus the offset, here 10. The advantage of this mode is that no changes are necessary if the program is moved in memory; the tenth byte following the contents of the program counter is always accessed.

```
002026  30C0            MOVE    D0,(A0)+
```

This addressing mode is very useful for working with tables. After the contents of D0 are stored at the address in A0, the contents of A0 are automatically incremented by 1, 2, or 4, depending on whether a byte, word, or long word was transferred. Afterwards, the next execution of the instruction automatically accesses the next table element.

```
002028  3100            MOVE    D0,-(A0)
```

This operation of this instruction is similar to the operation of the previous one. Here, however, the contents of the address register is decremented *prior* to execution of the instruction. Stack structures can be easily implemented with these two instructions.

```
00202A  32D8          MOVE     (A0)+,(A1)+
```

This example shows indirect addressing with post-incrementation of both operands. The contents of the memory location contained at address register A0 is moved to the memory location contained in address register A1. Afterwards, both registers A0 and A1 are incremented. This allows you to transfer entire memory blocks simply. Note that the contents of a memory block are transferred directly to the destination address without having to be loaded into a register in the 68000.

```
00202C  48E7F7F8      MOVEM.L  D0-D3/D5-D7/A0-A4,-(A7)
```

This instruction is unique to the 68000. Here the contents of registers D0-D3, D5-D7, and A0-A4 are placed on the stack with a single command. Address register A7 functions as the stack pointer. This instruction can replace up to 16 individual instructions for saving register contents on the stack, as is often required at the start of subroutines or interrupt service routines. The next instruction fetches the registers from the stack again.

```
002030 4CDF1FEF       MOVEM.L  (A7)+,D0-D3/D5-D7/A0-A4
```

In order to give a concrete demonstration of the capability of this processor, let's perform the same task first with a 6502 processor and then with a 68000. As an example, we will copy a 4K block of memory from address $3000 to $4000.

```
        ; 6502 EXAMPLE
3000          SOURCE  =  $3000
4000          DEST    =  $4000
1000          NUMBER  =  $1000   ; 4K BYTE
00F0          AD1     =  $F0     ; POINTER TO SOURCE
00F2          AD2     =  $F2     ; POINTER TO DEST
                      *= $2000
2000 A9 00            LDA  #<SOURCE
2002 85 F0            STA  AD1
2004 A9 30            LDA  #>SOURCE
2006 85 F1            STA  AD1+1   ; INITIALIZATION
2008 A9 00            LDA  #<DEST
200A 85 F2            STA  AD2
200C A9 40            LDA  #>DEST
200E 85 F3            STA  AD2+1
2010 A2 10            LDX  #NUMBER/256  ;OUTER LOOP COUNTER
2012 A0 00            LDY  #0      ; INNER LOOP COUNTER
2014 B1 F0 LOOP       LDA  (AD1),Y ; SOURCE BYTE
2016 91 F2            STA  (AD2),Y ; COPY TO DEST BYTE
2018 C8               INY
2019 D0 F9            BNE  LOOP
201B E6 F1            INC  AD1+1   ; INCREMENT POINTER
201D E6 F3            INC  AD2+1
201F CA               DEX          ; OUTER LOOP
2020 D0 F2            BNE  LOOP
```

Calculating the time that this 6502 program takes to execute, we find that it requires 65541 clock cycles. With a standard clock frequency of 1 MHz, this is 65.54 ms (milliseconds). Now let's "translate" the program into 68000 code.

```
003000          SOURCE =     $3000
004000          DEST   =     $4000
001000          NUMBER =     $1000

002000 207C00003000         MOVE.L  #SOURCE,A0     12
002006 227C00004000         MOVE.L  #DEST,A1       12
00200C 303C1000             MOVE.W  #NUMBER,D0      8
002010 12D8          LOOP   MOVE.B  (A0)+,(A1)+    12*4096
002012 51C8FFFC             DBRA    D0,LOOP        10*4096
                                          Total     90144
```

Even if you don't understand all of the details of the program, you can easily see that the 68000 program is shorter, in both the initialization and the actual program loop which performs the transfer. Since the 6502 only has an 8-bit register, a loop which is to be executed 4096 times is performed with two nested loops. Even loading the pointers with their starting values must be done in two steps on the 6502.

On the 68000, only one load command is required to initialize the registers for source, destination and number of iterations. The loop itself consists only of two instructions: the first copies a byte from the source to the destination address and automatically increments both pointers; the second decrements the loop counter and jumps back to the start of LOOP, as long as the counter D0 has not been decremented to zero.

The time to execute the individual instructions follows each instruction. We find that it takes 90144 clock cycles. With a 68000 driven at a standard clock frequency of 8 MHz, this takes 11.27 ms. This is already 5.8 times faster than the 6502. But we have not yet exhausted the power of the 68000. The above example transferred 4096 individual bytes.

But a 16-bit processor for such a task isn't very sensible. In the next example, we'll transfer 16-bit words in a single pass. This allows us to cut the number of loop iterations in half.

```
003000          SOURCE  =   $3000
004000          DEST    =   $4000
001000          NUMBER  =   $1000

002000 207C00003000        MOVE.L #SOURCE,A0       12
002006 227C00004000        MOVE.L #DEST,A1         12
00200C 303C0800            MOVE.W #NUMBER/2,D0      8
002010 32D8        LOOP    MOVE.W (A0)+,(A1)+      12*2048
002012 51C8FFFC            DBRA   D0,LOOP          10*2048
                                  Total   --->      45088
```

Now, we need only 45088 clock cycles or 5.64 ms to execute. This is about 11.6 times faster than the 6502. But the 68000 can work with long words, too. It naturally takes longer to transfer a long word than to transfer a word, but not as long as it takes to transfer two individual words. In addition, we can halve the number of loop iterations again. Here's the code:

```
003000          SOURCE  =   $3000
004000          DEST    =   $4000
001000          NUMBER  =   $1000

002000 41F83000              LEA.L  SOURCE,A0         8
002004 43F84000              LEA.L  DEST,A1           8
002008 303C0400              MOVE.W #NUMBER/4,D0      8
00200C 22D8          LOOP    MOVE.L (A0)+,(A1)+       20*1024
00200E 51C8FFFC              DBRA   D0,LOOP           10*1024
                             Total                     30744
```

Now we need only 30744 clock cycles or 3.84 ms. This makes our 68000 program about 17.1 times faster than the 6502. In this example we used the LEA (Load Effective Address) instruction for initializing the address registers. This instruction is faster than the MOVE instruction and also requires fewer bytes. The processing width of the instructions in these examples were indicated by appending ".B", ".W", ".L". In general, we can say that the 68000 is about 10 to 20 times faster in program execution than most popular 8-bit processors. In addition, the programs are usually shorter and easier to read thanks to the powerful and highly orthogonal instruction set. In special cases, the speed advantage over 8-bit processors can be still larger. This is the case when certain tasks must be performed where there are already commands on the 68000 which the 8-bit machines lack, such as multiplication or division.

1.4. ADVANCED FEATURES OF THE 68000

The 68000 can be operated in two modes. The first is the user mode and the second is the system or supervisor mode.

This permits a strict separation between the operating system and user programs. Some instructions are not allowed in the user mode--they are *privileged.* If an attempt is made to execute these instructions anyway, the 68000 branches to an exception handling routine similar to an interrupt. This routine, which is part of the operating system, can then react accordingly, by outputting a message, for instance. Privileged instructions include the STOP command which halts the processor, and the RESET command which resets the peripheral devices.

In general, these privileged instructions are reserved for operating system functions that protect important computer resources. For example, if valuable data is kept in a computer system, you may want to secure it from being read. If all of the facilities of the operating system are available to a user, then he may be able to overcome the security. But if some instructions are not available to him, then it is possible to keep him from accessing the data. This is one of the main purposes of privileged instructions.

As you can see from the register assignments, there is a different stack pointer, A7 for user and supervisor. They are referred to as the USP (user stack pointer) and SSP (system stack pointer). The operating system stack pointer cannot be changed by a user program, only from supervisory mode.

Similarly, interrupt priorities can be changed only from the supervisory mode. The 68000 has seven interrupt priorities. If the interrupt priority is zero, then all interrupts are permitted. Under the highest interrupt priority, only the non-maskable interrupt is permitted. If the interrupt mask is set to 3, for example, only interrupts 4 to 7 are permitted. If an interrupt occurs, the priority is automatically set to the value of this interruption during the execution. In this way, an active routine servicing an interrupt cannot be interrupted by the occurrence of one with lower or equal value. The processor is automatically switched to supervisory mode when an interrupt occurs.

Another feature of the 68000 is the CHK instruction. This instruction continuously checks the contents of an address register and if it falls outside of a desired address range, causes an interrupt. This is used in multi- user applications for example, to make sure that each user accesses only his assigned area of memory. If not, the CHK instruction can branch to an operating system routine which can then react accordingly.

Another use for this instruction is for high-level language compilers. Here, the instruction can be used to check if the index of an array remains within the declared limits and to output an error message if it does not.

The 68000 has a hardware feature which branches to an exception routine if an attempt is made to access an address range not installed on the computer. Since the 68000 can indicate if it is in the supervisor mode or user mode via the status lines, you can create a simple circuit if the processor attempts to address a certain area in user mode. This is done in the Atari ST. So it's not possible to access the stack area of the supervisor or the peripherals from the user mode. If an attempt is made, the bus error routine of the operating system is activated.

The 68000 has features to support high-level programming languages in which recursive calls are used. The LINK instruction can be used to reserve a data area to save the subroutine's local variables on the stack. The stack area can be freed again after subroutine is finished with UNLK instruction.

Since systems with a few hundred kilobytes of memory such as the Atari ST generally use only disk operating systems (DOS), the user programs are loaded into RAM. If several programs or program modules are combined, it can no longer be guaranteed that programs are always loaded at the address for which they were originally written. To do this one needs either a linker--a program which recalculates the absolute addresses of a program for another memory area--or a program which can run anywhere in memory. Such a program is said to be *relocatable*. The 68000 is ideally suited for writing such programs because it has program-counter relative instructions for subroutine calls, branches, and data accesses.

In conclusion, we can say that the 68000 is excellently suited for larger microcomputer systems with lots of memory. It supports the demands which are placed on such a system today. It offers separation between operating system and user programs. So it is well-suited for multi-user applications as well as for the implementation of modern high-level languages.

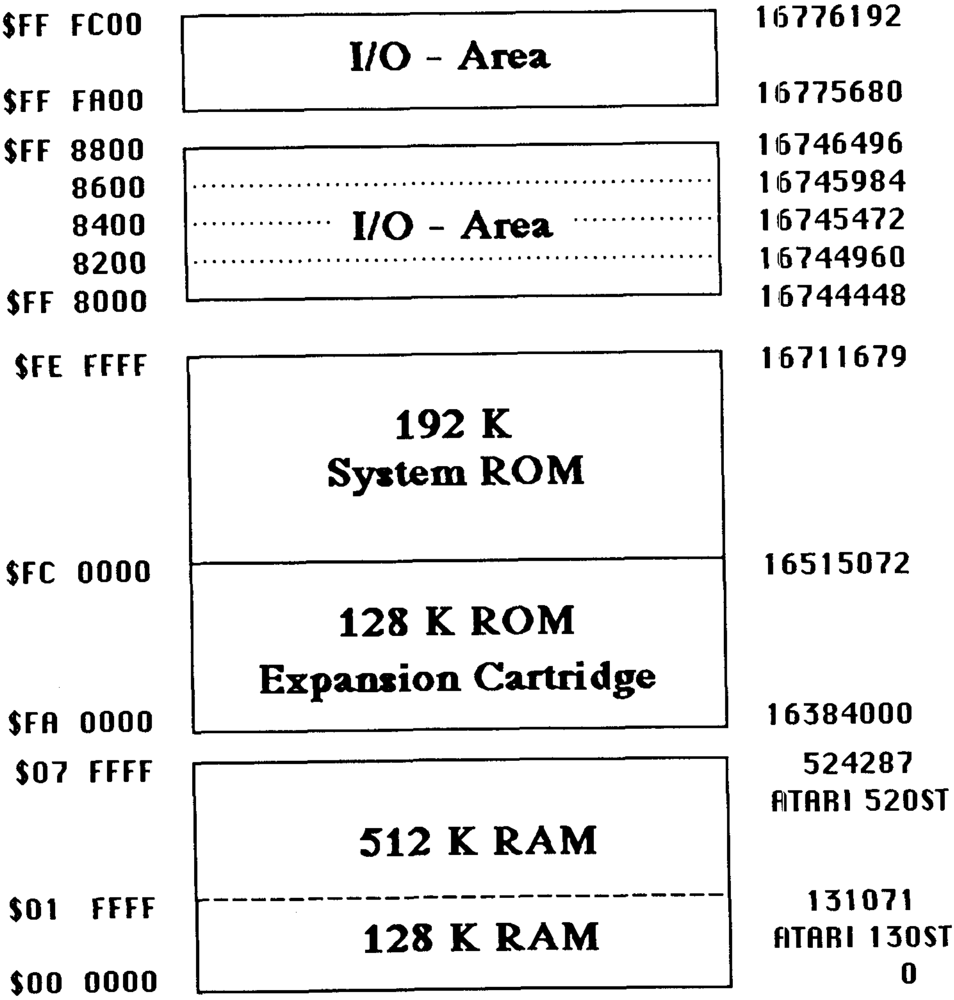
2. THE ARCHITECTURE OF THE ATARI ST

Now that we have learned a few fundamental things about the 68000 CPU, let's look at the layout of the Atari ST in more detail.

Let's begin with the ST's memory layout. As you already know, the 68000 can address 16 MB of memory. In the Atari ST 520, 512K or one-half megabyte of RAM is available. This is only 1/ 32nd of its available address range. The RAM area starts at address 0 and goes to address $07FFFF or decimal 524,287. The Atari ST 130 has 128K of RAM which ranges from address 0 to $01FFFF or decimal 131,071. One-half megabyte is eight times as much memory as an 8-bit computer with 64K. On the Atari ST, almost all of this memory is available to the user, as we will see later. On other 16-bit computers with 128K of RAM, more than half of the memory is often taken up by the operating system so that often no more than 32K is available for the user.

Not so with the Atari ST. Here, almost all of the operating system is in ROM. This ROM area is large, 192K, and lies from address $FC0000 to $FEFFFF (decimal 16,515,072 to 16,711,679). This ROM contains the BIOS, the actual operating system and GEM (more about this later). Only the scratch-pad memory needed by the operating system and the screen storage lie in RAM.

Fig. 2-1: MEMORY MAP



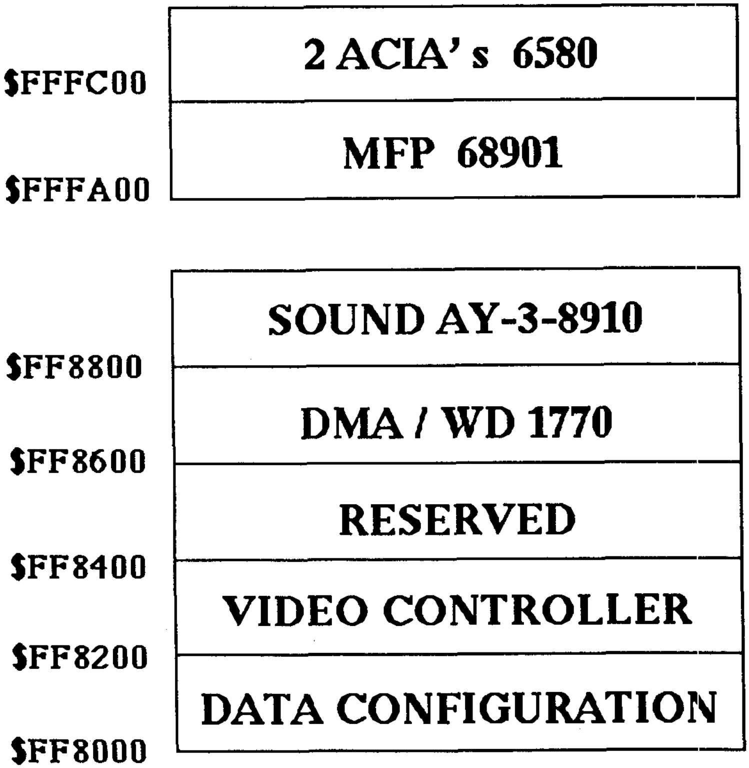
The fact that the operating system is built into ROM has several advantages. For one, the operating system does not have to be loaded from disk. For another, it does not occupy any RAM, which results in more available RAM for the user. With an address range of 16M, the space consumed by the ROM can be ignored since it is just one percent of the total address range allowable.

For those not satisfied with this, the address range from $FA0000 to $FBFFFF (decimal 16,384,000 to 16,515,071) is reserved for ROM cartridges. User programs as well as operating system expansions such as additional languages can be placed in this 128K range. The advantage over loading from disk is again clear: the loading procedure is avoided and no RAM is consumed.

The range from address $FF8000 (decimal 16,744,448) to the end of the physical address range is reserved for peripherals. 512 bytes is set aside for each device. This is a total of 32K for peripherals, 0.2% of the total address range. The diagram on the next page shows how the I/O area is divided.

Of the 128K or 512K (depending on the model), only 32K is required for the screen RAM. This RAM contains the bit map which serves for the representation of the picture on the monitor. The screen area can lie anywhere in RAM, thereby making it possible to switch between several screens, for instance.

Fig. 2-2: I/O ASSIGNMENTS



2.1. ATARI ST INTERFACES

On the next page is a block diagram of the Atari ST. Using this as a starting point, we'll examine the peripheral interfaces and devices.

As an overview, we first list the major interfaces:

* TWO 3 1/2" DISK DRIVES

* DMA INTERFACE FOR CONNECTION OF A HARD DISK WITH 10 MB STORAGE CAPACITY

* CENTRONICS PARALLEL INTERFACE FOR CONNECTION OF A PRINTER

* SERIAL RS 232 INTERFACE FOR CONNECTION OF DEVICES SUCH AS A MODEM OR A PRINTER

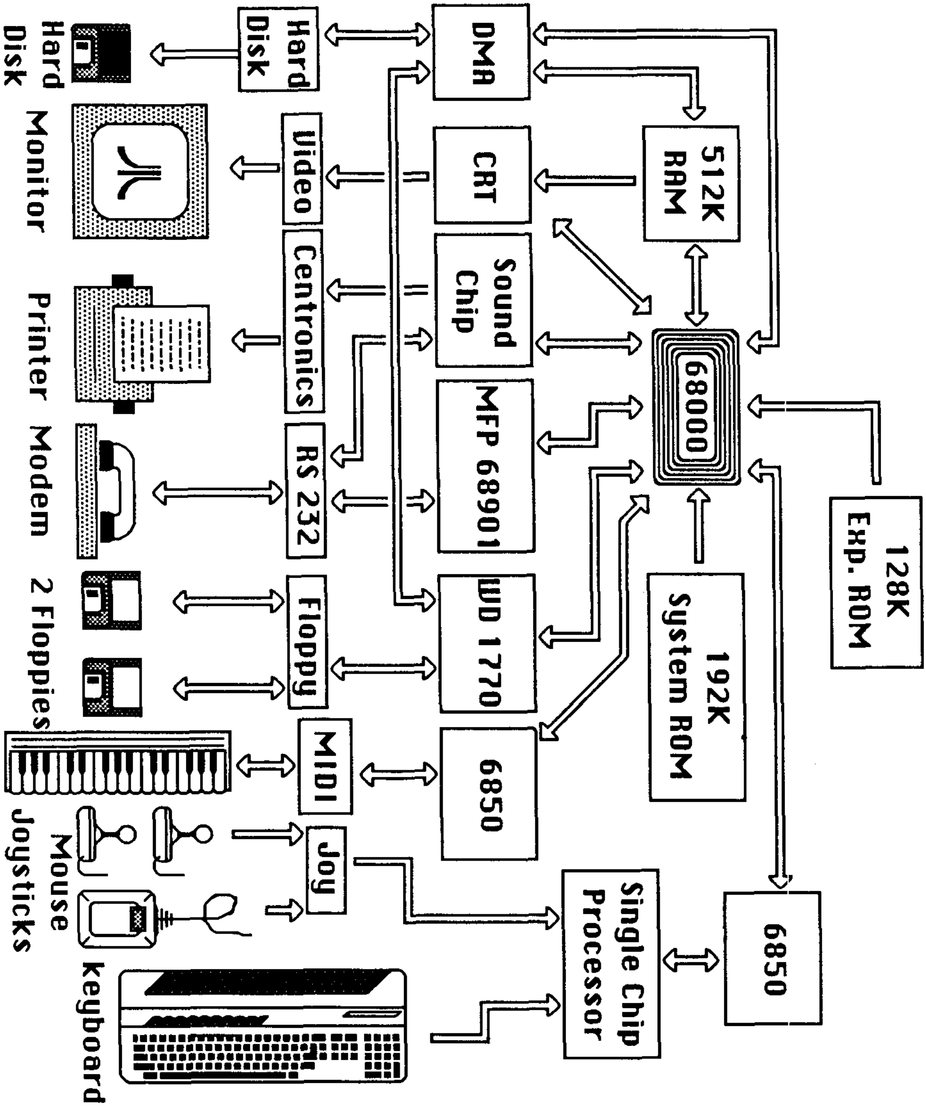
* MIDI INTERFACE FOR CONTROL OF EXTERNAL MUSIC SYNTHESIZERS

* 2 JOYSTICK PORTS, ONE FOR CONNECTION OF A MOUSE

* VIDEO CONNECTIONS FOR RF (TELEVISION), B/W MONITOR, AND RGB FOR COLOR MONITOR

* CONNECTION OF AN INTELLIGENT KEYBOARD

Fig. 2-3: BLOCK DIAGRAM of the ATARI ST



2.2. FLOPPY DISK INTERFACE

The most important peripheral device for a microcomputer is the mass storage device. The Atari ST allows up to two floppy disk drives to be connected, both of which use 3 1/2" diskettes. Atari uses the more compact 3 1/2" disk instead of the more common 5 1/4" disks. They offer several advantages. Even though they have about the same storage capacity, they are smaller than the 5 1/4" floppies. The diskettes are also less sensitive to mechanical damage since they are contained in a hard plastic case and so the magnetic surface is better protected. The opening for the read/write head on the diskette is also protected by a metal shutter which moves aside when the disk is inserted in the drive. In place of a write protect notch of the minifloppies, the 3 1/2" diskettes have a slide to protect the diskette from being written to. The smaller drives allow a smaller physical package and reduced cost.

The Atari ST has a built-in controller for operating the disk drives (WD 1770 from Western Digital). No special interface is necessary to connect the drives--a simple cable suffices. The disk controller receives its commands from the processor. The disk controller does not get the data to be written to and later read back in from the processor, but from a DMA component. The next section about the hard disk will further explain the DMA.

The disk controller is responsible for the way in which the individual bits are written to the diskette and for reading them in again later. The processor gives the disk controller commands such as "read sector" or "write sector". The disk controller can also format diskettes. The processor supplies the data which specifies the desired format of the diskette: how many tracks are used; how many sectors per track; how many bytes per sector; etc.

In addition, the disk controller is responsible for data integrity. To detect read errors, the disk controller creates a checksum called CRC (Cyclic Redundance Check) bytes, which are written to the disk following the data. Later, these CRC bytes are compared with the value calculated when the data is reread. If these two values are not identical, then the disk controller tells the processor via its status register that an error occurred.

The floppy disk has been the standard mass storage device for a microcomputer, and can be used for storing programs and data.

2.3. THE HARD DISK

If you work with an Atari ST 520 and a 500K disk drive, you'll know that the diskette has roughly the same capacity as the RAM capacity of the computer. If you use the computer for word processing, or you want to save entire graphic pages of 32K each on diskette, you quickly reach the storage capacity of the diskette. The hard disk is a way to overcome this limitation. Atari will offer a hard disk for the ST with 10 MB capacity.

What are the major differences between a hard disk and a floppy?

The most spectacular difference for the user is the enormous storage capacity. The 10 MB hard disk offered for the ATARI ST holds 20 to 40 times as much data as will fit onto a floppy. How is this possible?

In contrast to a floppy, a hard disk consists of a hard, rigid platter, usually made of aluminum and covered with a thin magnetic layer. The diameter of the platters on the Atari drives is the same as the floppys--3 1/2". The data can be written to the rotating platter or read back with a magnetic head, as with a floppy drive.

The essential difference between the hard disk and a floppy is that the hard platter rotates at 3000 RPM instead of 300 revolutions per minute.

Another difference is that the read/write head in the hard drive never touches the platter but "flies" over it at the very low altitude of half a micrometer (0.0005 mm). Such a method naturally requires a great deal of mechanical precision. A dust or smoke particle that comes between the head and the platter can damage the platter and result in subsequent data loss. For this reason, the platter is enclosed in a hermetically sealed case and cannot be changed like a normal diskette.

Coupled with higher quality components and increased speed of rotation, the track density of a hard disk is more than 800 tpi (tracks per inch) while the recording bit density is over 8000 bits per inch. The corresponding track density for a 3 1/2" floppy is typically 135 tpi.

As a result of the denser track layout, the hard disk also reduces the time the head needs to move from one track to another. This results in faster access time. A more important factor in performance is the data transfer rate. Because the platter rotates ten times faster than the floppy and is recorded at a higher density, the hard disk can deliver the data with a speed of about one megabyte per second. With a floppy, the data can be read at a speed of "only" about 25K per second.

In order to transfer data from a storage medium to the computer, or the other way around, peripheral interfaces are normally used to execute these tasks. To read external data, the processor reads the contents of the ports of the interface and then writes the value into memory. The I/O interface informs the processor via special registers when the next byte is ready and can be fetched by the processor. This method has proven itself and works quite well. Depending on the processor, transfer rates up to 100K per second can be obtained.

2.4. HIGH SPEED THROUGH DMA

In order to make full use of the transfer speed another method has be devised. The method is called "Direct Memory Access" or DMA for short. What's this all about? Our goal is to transfer data from mass storage to the memory of the computer. A way has been found to do this directly without involving the processor. There is a special component for this task, the DMA controller. This device has the job of bypassing the processor and writing the data coming from the hard disk directly into the memory of the computer. The DMA controller can also do the reverse when writing to the hard disk--the data is read from memory and sent directly to the hard disk.

The DMA controller is a custom I/O component for the processor. To send data to the hard disk, for instance, it must program the DMA controller appropriately. Among other things, the DMA controller must know the address range which is to be sent to the hard disk. When it has received the appropriate command, the DMA controller informs the processor via certain control lines that it now wants to access the data and address lines. The processor frees the bus and goes into a wait state. Now the DMA controller can access the entire memory range, read the data and send it to the hard disk. This occurs at the same high speed which the processor reads data from the memory. When the transfer is done, the DMA controller frees the bus and the processor can continue. On the Atari ST, a data transmission rate to and from the hard disk of 1.33 MB per second is possible. This allows the entire contents of the memory to be send to the hard disk in less than half a second!

For the sake of fairness we want to explain the disadvantages of a hard disk. For one, we cannot change the platter. This makes a hard disk unsuited for exchanging data between similar computers. The second and more important limitation involves the security of the data. If the hard disk should ever become defective, more than 10 MB of data may be destroyed. This can happen if a dust particle gets between the head and the platter, for instance. When something like this happens it is called a "head crash". In connection with the data integrity issue, there is often no simple way of backing the entire hard disk up. There are tape drives sold for this purpose, but they are frequently more expensive than the hard disk.

Since one usually works with one floppy and one hard disk in practice, the most important data should be copied from the hard disk to floppy diskettes at regular intervals. Hard disks today have quite a high quality standard and such errors rarely occur.

2.5. THE PRINTER INTERFACE

So that you can get things down in black and white, the Atari ST has a printer interface. With a printer you can reproduce documents. More interesting though is the possibility to output graphics as hardcopy. Because (as you will see) the screen of the Atari ST is always represented as graphics, the resolution in monochrome mode is 640 points horizontally by 400 points vertically, it is possible to output the contents of any desired screen to paper with a dot-matrix printer capable of graphics.

To connect a printer to a computer, both must have compatible interfaces. The Centronics parallel interface is considered the standard interface which printers and computers use. The Centronics interface is a parallel interface in which the data is transmitted byte by byte. Each character, such as a letter or digit, is represented in the computer with something called an ASCII code. In this code, each character is denoted through one byte or 8 bits. This makes it possible to represent a total of 256 different characters. For example, the letter A is represented by the ASCII code 65. When a character is to be transferred from the computer to the printer, the printer must be sent the corresponding byte of the ASCII code. With the Centronics interface, each bit has its own line and two handshake lines are used so that the computer and printer can agree on the time of the transfer.

If the computer wants to send a character to the printer, it places the data on the interface lines and puts a low signal on the STROBE line for a brief time (about one microsecond). This tells the printer that a character is ready and can be read from the interface. The BUSY line is required so that the computer knows when the printer has accepted the byte so it can send the next byte. As long as the printer is busy with the accepting and processing of a character, it makes the BUSY line high. The computer need only wait until the BUSY line goes back to low before it sends the next byte with a STROBE pulse.

Here are the lines and the data directions.

COMPUTER		PRINTER	
PSG PORT A6		STROBE	==>
PSG PORT B0		D0	==>
PSG PORT B1		D1	==>
PSG PORT B2		D2	==>
PSG PORT B3		D3	==>
PSG PORT B4		D4	==>
PSG PORT B5		D5	==>
PSG PORT B6		D6	==>
PSG PORT B7		D7	==>
MFP IO	<==	BUSY	
	GND		

PSG=Programmable Sound Genarator
MFP=Multi-Function Peripheral

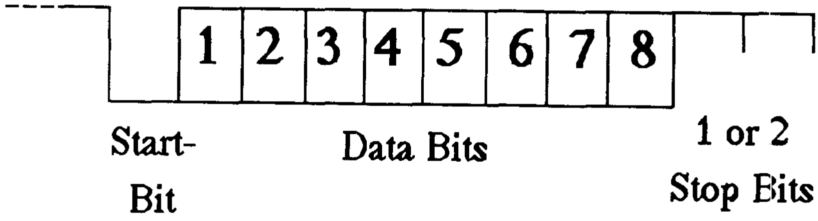
2.6. THE SERIAL INTERFACE

In contrast to parallel data transfer, which has a line available for each bit, only one line is required for the data with serial transmission. When a byte is to be transferred, the bits are transmitted one after the other over the same line. When sending such a bit stream, the receiver must know with which bit transmission of a new byte starts.

Two procedures have been developed to solve this problem. The first is called synchronous transmission. Here the sender and receiver are supplied with the same clock so that they, as the name implies, operate synchonously and can assign the bits received to the bytes properly. Since this method requires a coupling between sender and receiver, it is seldom used.

Much more widely used is the asynchronous method. Here the sender and receiver need only be connected to each other over one data line and a common ground line. This data line is at a specific condition in the wait state. If the sender wants to send a byte, it first sends a start bit by placing the data line low. Now the receiver knows that a data byte follows and can prepare to receive it. The individual bits are now transmitted over the data lines with low or high signals depending on their value. After the transfer the sender sends one or two stop bits, which at the same time correspond to the rest state. When the next byte is to be transferred the whole thing starts over from the beginning. The Figure 2-4 should clarify the matter.

Fig. 2-4: SERIAL TRANSMISSION



With this method any amount of time may pass between the individual bytes. A prerequisite of this procedure is that the receiver must constantly watch the data line so as not to miss the start bit. After a start bit is received it "collects" the data bits and puts them together into bytes. In order to inform the processor that a data byte has arrived, it generates an interrupt which causes the processor to get the complete byte from the peripheral component and write it to a buffer for later processing, for example. But let us take another look at the actual transmission.

The individual bits after the start bit have a set length. The sender and receiver must agree on this length before the transmission is attempted because the receiver has no way of determining this length. In our example, 8 bits are used. Specific values have been established for the times of each bit. The shorter the time for each bit, the greater the transmission speed. The speed is measured in bits per second. The name baud has been established for this.

BAUD RATE	BIT LENGTH (MS)
500	20.00
110	9.09
150	6.67
300	3.33
600	1.67
1200	0.83
1800	0.56
2400	0.42
3600	0.28
4800	0.21
7200	0.14
9600	0.10
19200	0.05

These are the baud rates at which the Atari ST can send data. While the Centronics interface is designed only for sending information, the serial interface is suited for receiving data as well. A second data line is used for this which the Atari ST uses as a serial data input. This input can be programmed for the same transmission speeds.

This serial transfer method is usually called the RS-232 standard. It functions faultlessly as long as the receiver processes the data as quickly as the sender transmits it. If the receiver is busy processing earlier data when the next byte arrives, the incoming data is lost without the sender even being aware. To prevent this loss of data several techniques termed handshaking are used. One method employs hardware and two additional lines, the "Clear To Send" or CTS line and "Request To Send" or RTS line. When the receiver is ready to receive a byte, it signals this through an appropriate signal on the RTS line. The sender first checks the line before making a transmission and waits as long as required until the receiver can accept the next byte. This procedure is often used when connecting terminals or printers. The disadvantage is that additional lines are required.

The Atari ST supports the following handshake lines for the RS-232 interface:

PIN	ABBR.	SIGNIFICANCE	I/O COMPONENT
4	RTS	REQUEST TO SEND	PSG PORT A3
5	CTS	CLEAR TO SEND	MFP I2
8	DCD	DATA CARRIER DETECT	MFP I1
20	DTR	DATA TERMINAL READY	PSG PORT A4
22	RI	RING INDICATOR	MFP I6

Another method which has been gaining more and more use is data transmission over the telephone. Here the individual bits are converted into different tones, transmitted over the phone lines, and converted back into voltage signals by the receiver. Devices which make such a conversion possible are called modems. When using the telephone, however, we have only one line running in each direction. The hardware method of handshaking described above is therefore impossible. A software method has been devised that allows handshaking over the phone lines. If the receiver wants to halt the transmission, it sends the character XOFF. This character has the ASCII code 19 and can be entered as a Control-S. Once the receiver is ready again to accept more data, it lets the sender know this by sending it XON which has the ASCII code 17 and can be entered as Control-Q.

The Atari ST supports the RS-232 interface through the operating system. The baud rate, number of data bits to be transmitted, and the type of handshaking can be programmed through the appropriate calls.You choose between no handshake, RTS/CTS, or XON/XOFF protocol. In addition it is possible to check the bytes transmitted with a parity bit.

2.7. THE MFP 68901

So far in this chapter we have talked about the peripheral components which are responsible for transferring data between the computer and the outside world. As an example of such chips, we want to take a closer look at the MFP 68901.

MFP is an abbreviation for "Multi-Function Peripheral". The name implies that this chip can perform several tasks. The MFP 68901 is a relatively new I/O component in the 68000 family.

These are the chip specifications:

*** 8 individually programmable input/output lines with interrupt capability**

*** Interrupt controller for 16 interrupt sources with individual masking**

*** Four timers, two of which have multiple functions**

*** The timers can be used as baud-rate generators for the serial channel**

*** One channel synchronous and asychronous serial input/output (USART, Universal Synchronous/Asynchronous Receiver/Transmitter)**

The registers of a peripheral component are addressed by the processor in exactly the same way as memory locations. The MFP 68901 has 24 internal registers, the significance of each is shown in TABLE 2-1:

Table 2-1: MFP 68901 INTERNAL REGISTERS

REGISTER	ABBR.	DESIGNATION
1	GPIP	GENERAL PURPOSE I/O REGISTER
3	AER	ACTIVE EDGE REGISTER
5	DDR	DATA DIRECTION REGISTER
7	IERA	INTERRUPT ENABLE REGISTER A
9	IERB	INTERRUPT ENABLE REGISTER B
11	IPRA	INTERRUPT PENDING REGISTER A
13	IPRB	INTERRUPT PENDING REGISTER B
15	ISRA	INTERRUPT IN-SERVICE REGISTER A
17	ISRB	INTERRUPT IN-SERVICE REGISTER B
19	IMRA	INTERRUPT MASK REGISTER A
21	IMRB	INTERRUPT MASK REGISTER B
23	VR	VECTOR REGISTER
25	TACR	TIMER A CONTROL REGISTER
27	TBCR	TIMER B CONTROL REGISTER
29	TCDCR	TIMER C AND D CONTROL REGISTER
31	TADR	TIMER A DATA REGISTER
33	TBDR	TIMER B DATA REGISTER
35	TCDR	TIMER C DATA REGISTER
37	TDDR	TIMER D DATA REGISTER
39	SCR	SYNCHRONOUS CHAR. REGISTER
41	UCR	USART CONTROL REGISTER
43	RSR	RECEIVER STATUS REGISTER
45	TSR	TRANSMITTER STATUS REGISTER
47	UDR	USART DATA REGISTER

Since the MFP has an 8-bit data bus, the individual registers are numbered by two's. With the help of these registers, the processor can control the operating mode of the MFP, supply it with data to be output, and get input data from it.

A peculiarity of the MFP 68901 is the 16-channel interrupt controller. It can manage 16 interrupt sources and permit or mask (prohibit) them individually. These interrupts are prioritized, meaning that the interrupt sources have varying weights. The MFP 68901 can generate a unique vector number for each of these interrupts, so a separate interrupt service routine is available in the 68000 for each of the interrupts. The interrupt priorities are all the same as far as the 68000 is concerned; the prioritizing is taken care of in the 68901. The 68901 uses the highest level of maskable interrupt in the 68000, namely six.

The vertical sync interrupt resides at interrupt level four and the horizontal sync interrupt is at level two. These interrupts work as auto-vector interrupts which means that the vector number is determined by the 68000. The Atari ST uses only the top two of the three interrupt-level inputs, resulting in the fact that only levels 2, 4, and 6 can be used. Level 7, with the highest priority represents the NMI (non-maskable interrupt).

The MFP 68901 can manage 8 internal and 8 external interrupt sources which have the following priority:

Table 2-2: MFP68901 INTERRUPT PRIORITY

PRIORITY	CHANNEL	DESCRIPTION	
HIGHEST	15	EXTERNAL INTERRUPT 7	I7
	14	EXTERNAL INTERRUPT 6	I6
	13	TIMER A	
	12	RECEIVER BUFFER FULL	
	11	RECEIVER ERROR	
	10	SEND BUFFER EMPTY	
	9	TRANSMISSION ERROR	
	8	TIMER B	
	7	EXTERNAL INTERRUPT 5	I5
	6	EXTERNAL INTERRUPT 4	I4
	5	TIMER C	
	4	TIMER D	
	3	EXTERNAL INTERRUPT 3	I3
	2	EXTERNAL INTERRUPT 2	I2
	1	EXTERNAL INTERRUPT 1	I1
LOWEST	0	EXTERNAL INTERRUPT 0	I0

Each of these interrupt sources can be enabled or disabled by programming the MFP 68901. The eight external interrupt sources use the data lines of the 8-bit port as inputs. If all of the lines are not used as interrupt inputs, the rest can be used as normal port lines. Internal interrupt sources can be the timers A through D, which can be used in different operating modes. They can create delay times, measure pulse widths, generate square waves, or count events.

In addition, two of the timers can be used as baud rate generators for serial transmission. The 68901 can generate an interrupt when an entire byte has arrived serially. The byte can then be input from the USART data register by an appropriate routine. The MFP 68901 can call an interrupt routine, using interrupt channel 10, which places the next byte at its disposal when it has output a complete byte.

The MFP 68901 is housed in a 48-pin case and is connected to the asychronous bus of the 68000. The timers can be supplied with a clock independent of the clock frequency of the processor. In the Atari ST the 68901 serves as the serial interface and works as interrupt controller. The 16 interrupt levels of the 68901 are assigned as follows:

Table 2-3: INTERRUPT STRUCTURE OF ATARI ST

PRIORITY	CHANNEL	SIGNIFICANCE
HIGHEST	15	I7 MONOCHROME MONITOR DETECT
	14	I6 RS-232 RING INDICATOR
	13	TIMER A, SYSTEM CLOCK
	12	RS 232 RECEIVER BUFFER FULL
	11	RS 232 RECEIVER ERROR
	10	RS 232 SENDER BUFFER EMPTY
	9	RS 232 TRANSMISSION ERROR
	8	TIMER B, HORIZONTAL LINE RETURN
	7	I5 FLOPPY DISK CONTROLLER
	6	I4 ACIA 6850, KEYBOARD AND MIDI
	5	TIMER C
	4	TIMER D, RS 232 BAUD RATE GENERATOR
	3	I3 GPU OPERATION DONE
	2	I2 RS 232 CLEAR TO SEND
	1	I1 RS 232 DATA CARRIER DETECT
LOWEST	0	I0 CENTRONICS BUSY

2.8. THE SOUND CAPABILITIES OF THE ATARI ST

The Atari ST contains a built-in sound generator of type AY-3-8910 from General Instruments or a replacement type YM 2149 from Yamaha.

The 40-pin chip has the following specifications:

*** three independently programmable tone generators**

*** a programmable noise generator**

*** outputs which are completely software controlled**

*** programmable mixer for tone and noise**

*** 15 logarithmic volume levels**

*** programmable waveforms**

*** two bi-directional 8-bit ports**

The programmable sound generator AY-3-8910, abbreviated to PSG, is constructed as follows:

Three tone generators A, B, and C create the square waves for each channel. A noise generator produces a square wave with a random pulse width. Three mixers can mix the outputs of the tone generators with the noise generator.The D/A converter can be controlled with either the amplitude control or the waveform generator. The programmable wave form generator can modulate the tone outputs into different wave forms. The D/A converter creates 16 different volumes determined by the amplitude control.

The PSG has 16 registers which can be programmed by the processor:

Registers R0 through R5 serve to determine the pitch of the three channels A, B, and C. Registers R0 and R1 control channel A, R2 and R3 are responsible for channel B, and R4 and R5 for channel C. The value for the period of the tone is written to this registers as a multiple of the clock frequency of the PSG divided by PSG. Since the PSG in the Atari ST is driven at 2 MHz, the basic value for the period duration is 8 microseconds. Only twelve bits of the 16 bits in the two registers can be used; the top four bits are ignored. What frequencies can be created? We can write values between 1 and 4095 to the registers and we get the following values:

1 * 8 = 8 microseconds => 125,000 Hz
4095 * 8 = 32760 microseconds => 30 Hz

We can therefore create frequencies which far exceed the audible range.

Register R6 sets the basic frequency of the noise generator. Here only the 5 lowest bits are used.

Register R7 selects the signals of the tone generators and noise generator. With six bits you can specify which channels of the tone generator are to be switched to the corresponding outputs and to which channels the noise generator are to be be added. Bits 6 and 7 are responsible for the data direction of the two 8-bit ports.

Registers 8 thru 10 are the amplitude control. Registers 11 and 12 are used for programming the waveform. The remaining registers are used for the I/O ports. The 8-bit port B of the PSG (register 15) is used in the Atari ST for the Centronics interface. Port A (register 13), which can only be used as output, delivers the handshake signals for the serial and parallel interfaces. Three bits of port A are also used for the floppy interface to select the drives and heads.

ADSR control is possible with the programmable waveform. This is an abbreviation for Attack-Decay-Sustain-Release. This allows the attack time, decay time, sustain time, and release time of a tone to be set.

The Atari ST operating system contains procedures for passing the appropriate parameters to the PSG.

2.9. THE MIDI INTERFACE

Another feature of the Atari ST is the built-in MIDI interface. MIDI is the abbreviation for "Musical Instrument Digital Interface" and is a standardized interface for synthesizers, sequencers, home computers, rhythm devices and so on. Devices equipped with this interface can be operated together. Thus it is possible to control synthesizers with a MIDI interface from a computer such as the Atari ST.

From a hardware standpoint, the MIDI interface consists of a sender and a receiver for asynchronous serial transmission similar to the RS-232 interface. The MIDI transmission takes place at 31.25Kbaud with one start bit, eight data bits and one stop bit. This device therefore requires 320 microseconds to transfer these 10 bits, which corresponds to 3125 bytes per second. This high data rate is required so that real-time applications are possible.

Communication over the MIDI interface takes place with multi-byte messages. The first byte is a status byte. The status byte is denoted by a set seventh bit. The lower 4 bits select one of 16 channels to which the message applies. It is also possible to send messages to all connected devices.

The Atari ST has connections for MIDI IN and MIDI OUT for receiving and sending MIDI information. The optional MIDI THRU connection can be performed through software.

Operating system procedures are available to service the MIDI interface. These functions can output data to and read data from the MIDI interface.

The actual data input and output is interrupt controlled and is carried out by a 6850 ACIA circuit (Asychronous Communications Interface Adapter). Another operating system function returns the status of the MIDI interface. This allows us to determine if data is present at the MIDI interface or if it is ready for data. An operating system call can also be used to set the size and address of a buffer for temporary storage of the MIDI data received.

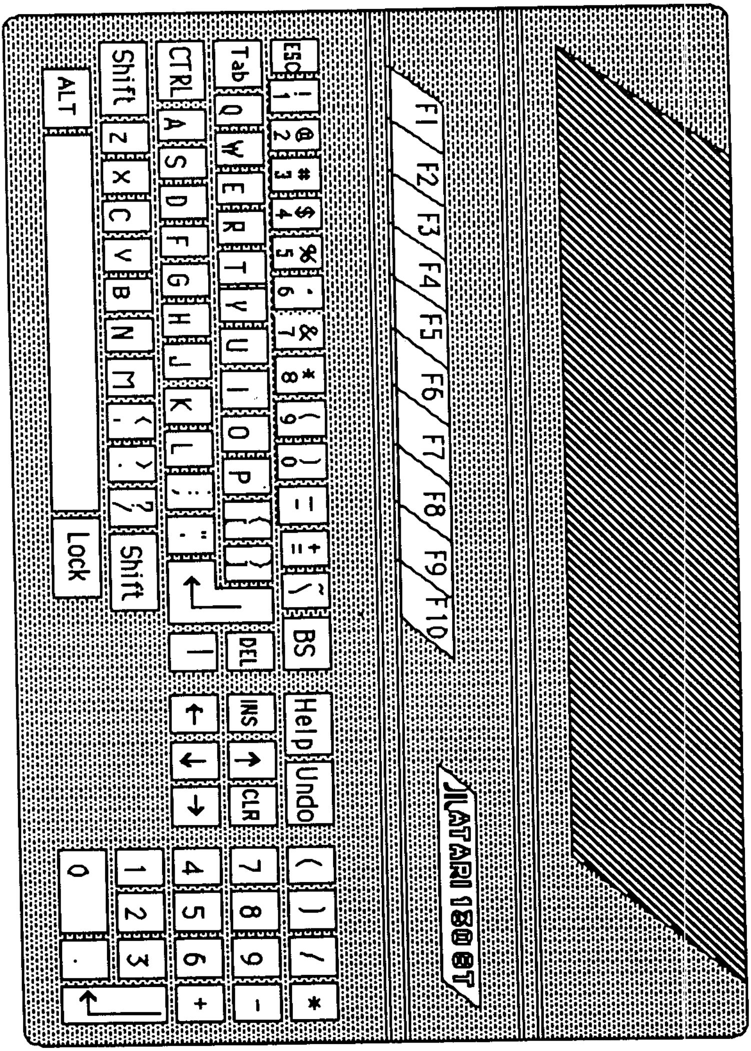
2.10. THE ATARI ST KEYBOARD

A serious computer needs a solid and reliable keyboard. Factors that determine the suitability of a keyboard include the number of keys, the layout of the keys and the ergonometric organization of the keyboard. The keyboard must not be too high and must be at the proper angle. Only when these things are correct is it possible to work at the computer for a long time without fatigue.

The keyboard on the Atari ST is divided into four sections. The first and largest section is a typewriter keyboard. Two large shift keys switch between lower and upper case or special characters. The control key in combination with other keys creates the non-printable control characters from 0 to 31. There is also an ESC (escape) key and an ALTernate key. This makes it possible to create any desired ASCII code. After the ALT key is depressed, three digits representing the ASCII code may be entered. The upper case letters can also be obtained without the shift keys by using the CAPS LOCK key.

The second section of the keyboard is the cursor pad. This consists of four cursor-control keys arranged in a T- shape. The HOME/CLEAR key places the cursor in the upper left-hand corner of the screen or, if used with the shift key, also erases the screen. A character can be inserted at the current cursor position with the "INSERT" key. The keys "HELP" and "UNDO" are also part of this section.

Fig. 2-5: KEYBOARD LAYOUT


F1 F2 F3 F4 F5 F6 F7 F8 F9 F10
Esc ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 - = + ~ BS Help Undo ( ) / *
Tab Q W E R T Y U I O P [ ] DEL INS CLR 7 8 9 -
CTRL A S D F G H J K L ; : ' " 4 5 6 +
Shift Z X C V B N M < , > . ? / Shift 1 2 3
ALT Lock 0 .
ATARI 1040ST

The numeric keypad is located on the right side of the computer. It allows fast entry of numerical data. In addition to the numerals and decimal point, the numeric keypad also has the keys "+", "-", "*", and "/" for the standard arithmetic functions, "(" and ")" and an "ENTER" key for terminating the input.

The Atari ST has ten function keys designated F1 through F10 located above the typewriter section. These keys can be assigned special purposes through software.

From a hardware standpoint, the keyboard actually appears to be an "intelligent" peripheral to the 68000 processor. The keyboard has a single-chip microcontroller, the HD63P01M1 from Hitachi, which has 29 I/O lines, a 16-bit timer and a serial interface built in. This CMOS processor also contains 128 bytes of RAM on the chip. In the prototype of the Atari ST, this processor has a 4K "piggy back" EPROM which contains the operating system for the keyboard processor. In production versions, the processor with a built-in ROM (HD63P01V1) will replace the EPROM.

This single-chip controller frees the 68000 from the task of decoding the keyboard matrix. It is well suited for this task since its has 29 I/O lines.

Communication from the HD63P01M1 takes place over a serial interface already intergrated into the controller. The 68000 uses a 6850 ACIA (Asynchronous Communications Interface Adapter) as its peripheral component. The data transmission takes place at a rate of 7800 baud. Because the serial interface is designed for both input and output, you can "program" the keyboard by sending data to the controller to perform such functions as CAPS LOCK.

The advantage of an intelligent keyboard lies in the fact that special functions can be achieved easily without having to burden the main processor. The 68000 gets the code of the key which is pressed just by requesting it from the controller.

This single-chip controller also handles input from the joysticks and/or mouse. The I/O lines that are not used for polling the keyboard are used for this. Two connectors on the right side of the Atari ST are used to attach the joysticks or the mouse. One port is used for the mouse. The four directions and the fire button on the joystick can be polled while the mouse sends the direction of movement or button press. The single-chip processor sends the keyboard, joystick, and mouse data to the 68000 via the serial interface. Since the data causes an interrupt at the 6850, it is made immediately available to the 68000. The character can then be read from the single-chip controller by the interrupt service routine.

The HD63P01M1 is an eight-bit CMOS processor which can be equipped with a 4 or 8K EPROM. It has an instruction set which is upward compatible to the 6800 and 6801. In addition, it has bit-oriented instructions which are useful for polling individual I/O lines.

2.11. THE ATARI ST VIDEO INTERFACE

Since the Atari ST does not have a built-in monitor, it must be connected to a video display terminal for operation. The simplest output is to connect the ST to a colour or black-and-white television set. The Atari ST has a built-in RF modulator to produce the necessary signals for the television receiver. But because of its limited screen resolution, a television set is not the optimal output device. Let's look at the other options for screen display.

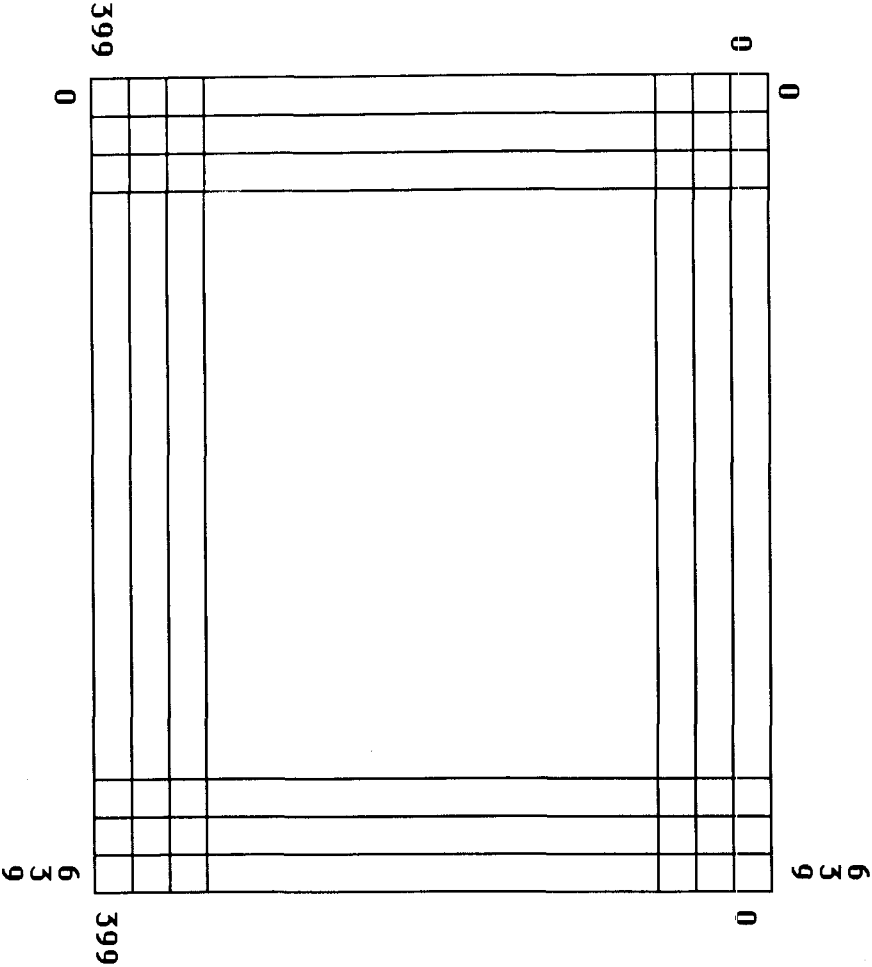
The Atari ST uses only the graphics mode for text and graphics output.

When creating a video picture, the screen is divided into individual points which can be separately set or cleared. On the screen, a set point appears white and a cleared point black. This assignment can be changed, however. Different colours can be assigned to these two states, for example.

For the sake of simplicity, let's start with the monochrome mode on the Atari ST. In this mode, the screen resolution is 640 points horizontally by 400 points vertically. This gives a total of 256,000 points which can be set or cleared independently of each other.

Each screen point can have one of two different states. In memory, one bit also has two possible conditions: zero or one. This suggests that each screen point be assigned to a bit in memory. In order to achieve the resolution mentioned before, we need 256,000 bits or 32K bytes. This is

Fig. 2-6: 640 x 400 BIT-MAPPED GRAPHICS



referred to as bit-mapped graphics. This 32K must be placed somewhere in RAM. With 512K of RAM, however, this is only one sixteenth of the total RAM.

The ST can do more than monochrome display. In addition to the high-resolution single-color mode with 640x400 screen points, the ST has another mode in which four colors can be displayed simultaneously on the screen. The resolution in this mode is now 640x200 points. How can we display four different colors with only half as many points? To answer this, we must consider how the colors are stored in the 32K of graphics RAM.

When one bit is used for a screen point, this bit can take on one of only two conditions, just as the corresponding screen point can be either set or unset. In the colour mode, these two bits are used for representing a single screen point. These bits can contain four different values.

0 0	=>	0
0 1	=>	1
1 0	=>	2
1 1	=>	3

A colour can be assigned to each of these four values. The video controller is responsible for the assignment and creation of the screen signals as is often referred to as a CRTC (Cathode-Ray Tube Controller).

In addition to the four-colour mode which provides 128,000 points of resolution, there is a multi-colour mode in which it is possible to display 16 different colours at the same time. By simply extending the method used for the four-colour mode, we can use four bits per point to represent 16 colours:

0000 => 0

0001 => 1

0010 => 2

0011 => 3

0100 => 4

0101 => 5

0110 => 6

0111 => 7

1000 => 8

1001 => 9

1010 => 10

1011 => 11

1100 => 12

1101 => 13

1110 => 14

1111 => 15

Since we have 256,000 bits of memory and we need four bits for each point, we can control 64,000 points. This gives us a resolution of 320 points horizontally and 200 vertically.

So the ST can display either four or sixteen colours at a time. These colours may be any one of 512 colours. These colours are combinations of the primary colours red, green and blue. Each of these primary colours can be used in one of eight brightness levels. This results in 8 * 8 * 8 = 512 combinations. With these colours available, practically all colour gradations can be represented. White results from of a superimposition of all three primary colours. If the primary colours are used only in brightness levels, corresponding grey levels are produced. The combination of red and green gives yellow, red and blue make purple (magenta), and blue and green make cyan. All other colours can also be formed with a partial mix of two primary colours.

To achieve the best picture quality, you should use the RGB output. RGB is an abbreviation for RedGreenBlue. With this interface, the signals for the individual colors are available separately and can be used directly for controlling the cathode rays in the monitor. The Atari ST has such an RGB interface for direct connection of an RGB monitor.

In order to connect a simpler monitor, the ST also has a composite video interface. Here, the three colour signals are mixed together into one video signal. The monitor later separates this signal back into the three colours, but this results in a loss of picture quality when compared to the RGB output. Another loss in quality happens if you connect it to a television set through the RF output. Here the video signal is first modulated into a high-frequency signal and then demodulated in the receiver. The signal must be further divided into the three primary colours.

In addition, televisions are not designed to display pictures with as high a resolution as the Atari ST generates.

The Atari ST also has a connection for a high- resolution monochrome monitor in order to make full use of the high-resolution in the single-colour mode.

With its four video interfaces, the Atari ST offers something for differing tastes in quality and price range. Atari will offer a high-resolution 12" monochrome and a 12" RGB color monitor for the ST.

Figures 2.7 and 2.8 show screen photos of the Atari ST which demonstrates the high resolution of 640x400 points.

Fig. 2-7: HI-RES ATARI ST SCREEN PHOTO

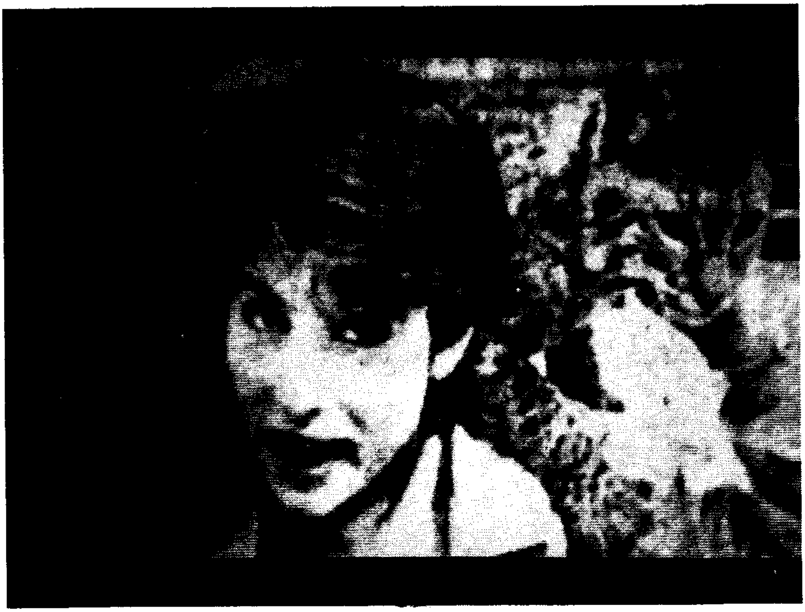
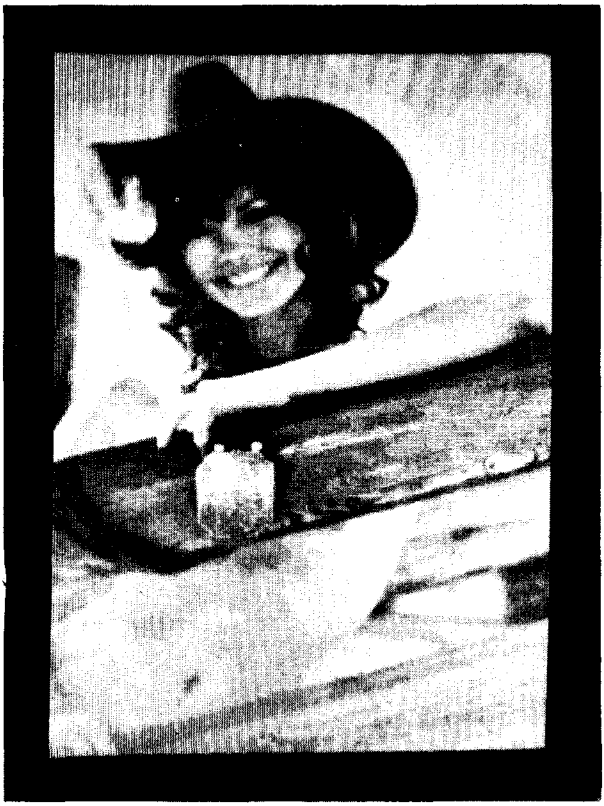


Fig. 2-8: HI-RES ATARI ST SCREEN PHOTO



3. THE ATARI ST OPERATING SYSTEM

Now that we've become a bit more acquainted with the hardware of the Atari ST, let's turn now to the operating system. First let's clarify what an operating system is, what tasks it has to perform, and what demands are placed on a modern operating system.

The capable hardware of the Atari ST is of no use without a program to direct the processor and peripheral components. The operating system is the "director". The operating system makes it possible for the computer to recognize a press on a key; to construct a command from multiple key presses; to carry out the commands that are entered; to send data to and receive data from peripheral devices that result from these commands and other functions as well.

For any computer, the primary input and output devices are the keyboard and the display screen. They make it possible to work with the computer interactively - the computer responds immediately to keyboard input with results on the screen. If the results are to be documented, the output can be routed to a printer. The operating system controls the printer so that it produces the desired results.

The Atari ST can support two joysticks and a mouse as additional input devices. The mouse is an input device which can simplify the input of data between the user and computer. To make such devices easy to use, the operating system must provide procedures to service these devices.

In order to exchange data with other computers, you can use a telephone, a modem, and a computer with a serial interface. Using this configuration you can connect to almost any computer in the world. These may be simple home computers or complex mainframes which manage huge data banks. Controlling the ST's serial interface also falls to the operating system.

The disk drive is a necessity for most computers. Recently hard disk drives, such as the 3 1/2" drives from Atari, have become available at a price which makes them a reasonable alternative to floppy drives. In most instances, the operating system is contained in the computer's memory while user programs such as word processing, data management, or high-level programming languages are kept on the mass storage and loaded into memory by the operating system when needed. After the program is loaded, the operating system passes control to the user program. Later, when the program is done, it returns control back to the operating system.

An important task of the operating system is that of moving programs and data between the computer and the mass storage. Often the operating system itself is stored on the mass storage device and is loaded into the computer when it is turned on. The name "Disk Operating System", or DOS for short, is used to name an operating system whose main task is the management of the disk drive.

But an operating system does more than simply load programs from the mass storage devices and then start them. It must also perform the input and output operations for user programs.

In the hardware description of the Atari ST we became acquainted with the peripheral components responsible for data input and output. If a user wishes to output text to a printer from within a program, the computer must know exactly which peripherial component the printer is connected to. Even if this is known, it is not enough to simply send the data to the peripheral component; the data that is output to the printer must also follow a certain *protocol.* In the description of the printer interface we saw that this is done with the STROBE and BUSY lines.

In general, a protocol is a set of rules which determines how data is sent between two devices. There is a protocol for serial interfaces too, such as the XON/XOFF protocol described earlier. Since input and output is performed by every program, it doesn't make sense for the user to program these things over and over again for each program. Furthermore, to program at this level requires that the programmer have a very detailed knowledge of both hardware and machine language.

So the operating system performs most of these tasks. The operating system contains ready-to-use routines called input/output routines. By using these routines you can output text to the screen, read characters from the keyboard, send or receive data over the serial interface, produce hardcopy on the printer or read data from or write data to a diskette.

To use these functions, the user simply calls the appropriate operating system routine. Since the operating system usually has to input or output data, the user must inform the operating system the location of the data and the type of transfer that is desired.

How should an operating system routine be designed?

Before answering this question, let's pause a few minutes to talk about the early days of small computers.

When the first personal computer appeared on the market ten years ago, its price was determined largely by the cost of the hardware. The cost of memory for example, was relatively expensive. This was the reason that most of the early computer had small amounts of memory - either 4K or 8K of RAM. Many users couldn't afford to buy more. But advances in production techniques and volume sales have reduced the cost per kilobyte to less than 20 pence. Within a year 1M (million) bit chips will most likely be available and most personal computers will have at least 1M-bytes of RAM. By replacing 256K-bit chips with 1M-bit chips the Atari ST can reach its 2M-byte memory limit.

Similarly, advances have been made in the manufacturing of mass storage devices that have reduced the prices of both floppy and hard disk drives.

In the past, the cost of the hardware components accounted for 80 to 90% of the price of a personal computer. As a result of this tremendous drop in component cost, the prices of the more recent computers have been steadily sinking.

There are some differences between the software and hardware costs. Developing software has been and still is labour intensive. As such, the many man-months or man-years that it takes to develop complex software is expensive. In order to maximize the software development investment, most developers try to design their software so that it can run on as many computers as possible.

One operating system which was developed with the idea of being used on many different computers, is called CP/M (Control Program for Microcomputers). Developed by a company named Digital Research to work on the Intel 8080 processor, it was later adapted to work with the Zilog Z-80 processor. It is written in such a way as to be able to work on computers developed by different manufacturers.

3.1 THE CP/M OPERATING SYSTEM

To make it easy to adapt one operating system to the differing computer hardware, CP/M was divided into three software components, each functionally separate from one another. These components are:

BIOS	BASIC INPUT OUTPUT SYSTEM
BDOS	BASIC DISK OPERATING SYSTEM
CCP	CONSOLE COMMAND PROCESSOR

The BIOS, or basic input output system, performs the actual interfacing to the hardware. This part of the operating system includes the routines for screen output, reading the keyboard, printer output and reading and writing individual sectors on floppy or hard disk.

The BIOS is hardware specific and must be rewritten to implement CP/M on a new computer. The BIOS has a preset number of simple input and output operations to perform. At the beginning of the BIOS program is a *jump table* for all of its operations. This table contains the memory location of the routine which performs a given operation. The routines themselves are adapted specifically for each different computer.

The BDOS, or basic disk operating system, is responsible for the logical organization of mass storage data. It manages data at the file level.

One of its main jobs is to manage the disk storage space. When you create a new file, for example, the BDOS searches for unused space on the disk. It selects the track and sector at which the data is stored. The user only needs to know the name of the file. To help manage the disk space, the BDOS use a directory containing the file names, type, size and location of the file. So when you later want to access the data in the file, the BDOS knows where to find it.

To reiterate, the BIOS is in charge of the physical management of the peripheral devices while the BDOS is responsible for the logical management of the disk drive. With help from the BDOS, programs and data can be stored on the disk and read back again.

The third component of the CP/M operating system, is the CCP or console command processor. It is the part of the operating system that is responsible for handling the main line of communication between the user and operating system from the keyboard. When CP/M starts up, the CCP responds with the following output on the screen:

```
A>
```

The A means that the first disk drive is selected. Under CP/M the drives are designated with letters which range from A to P. Therefore it is possible to connect up to 15 drives. The character > indicates that the operating system is now waiting for the user to input a command. We can now send a command to the CCP. To do this, we type the command on the keyboard. The characters are echoed on the screen as the user types them. When a command has been entered and terminated by pressing the RETURN-key, the CCP analyzes it.

Using CP/M, there are two types of commands: resident and non-resident. Resident commands are part of the operating system and are always available for use in memory.

The CCP recognizes the following resident commands:

DIR

This command displays the directory screen. The directory is a listing of the files stored on the selected disk drive. To display the directory of a different disk drive, add the letter designation of the drive followed by a colon. For example:

DIR B:

This displays the directory of drive B on the screen.

TYPE *fname*

This command displays a text file, named *fname*, on the screen.

TYPE TEXT.TXT

This displays the file TEXT.TXT. In CP/M, filenames consist of two parts separated by a . (period). The first part of the name may contain up to eight characters. The second part of the name, called a file type or extension, may contain up to three characters.

ERA fname

This command deletes (erases) a file from the disk. The name of the file to be deleted is **fname.**

ERA TEXT.TXT

This deletes the file TEXT.TXT.

CP/M also offers the option of erasing several files at once. To do this, you use wildcard characters. By replacing a character in *fname* with ?, you are referring to all filenames which match the remaining characters and which have any character in place of the ?. For example, TEXT?.TXT can refer to the following files:

TEXT1.TXT
TEXT2.TXT
TEXTE.TXT

The ? can be used in the first part of the name as well as the file type or extension. A second wildcard character is * (asterisk). The * matches all subsequent characters. So a file designation of **TEXT.*** refers to all files with a first part of TEXT and any extension. The question mark and asterisk can be combined, for example:

TEXT?.*

This refers to all files whose first part start with **TEXT**, are five characters long and may have any extension. If we enter the command:

ERA *.*

all files on the disk will be erased.

REN ***nfname=ofname***

This command renames an existing file from *ofname* to *nfname*. To rename our TEXT.TXT file to TEXTNEW.TXT, we can use the following command:

```
REN TEXTNEW.TXT=TEXT.TXT
```

USER ***uarea***

This command selects from among the different *user areas* on a floppy or hard disk. If you have a large capacity floppy or hard disk, you can group your files together in one *user area.*

USER 1

This command selects user area 1. An advantage of user areas is that a directory listing will show only those files in a particular user area. They may also be used to separate various application areas from each other. So when working with a hard disk which may contain dozens of files, you have immediate access to only those which are needed for a specific purpose.

SAVE blks fname

This command saves the current contents of memory to a file. With CP/M 2.2, this memory starts at address 100H (hexadecimal), through the start of the operating system which lies at the upper end of the memory. The parameter *blks* is the number of 256-byte blocks to be written to the file whose name is *fname*.

SAVE 10 NAME.EXT

This command writes 10 256-byte blocks to disk with a file named **NAME.EXT**. The range of memory written is from 0100 hex to 0B00 hex.

As previously mentioned, the CCP also recognizes non-resident commands. If you enter a command at the keyboard which is not one of the resident commands, then the operating system looks for a file on the disk with this name. The file must have an extension of **.COM**. For example, if you enter:

testcmd

the CCP looks for a file with the name **testcmd.COM**. If it finds one, then the program is loaded into memory and starts as if it were a resident command.

Part of the CP/M operating system includes several non-resident commands. The most important of these are:

STAT

The STAT command displays and permits altering the disk or file parameters and the logical assignment of input and output devices.

PIP

The PIP command (peripheral interchange program) is a universal copy and input/output program. Using PIP, you can copy files from one drive to another, combine several files into one, output files on the screen or printer or transfer data from a serial interface to a file. The data can be manipulated during the transfer. The line and page format of text data output to a printer can be determined, for example, or the output lines can be numbered.

ED

The ED command is a simple line-oriented editor for creating and editing text files. It can be used to create source files for assemblers and compilers, for example.

ASM

The ASM command is the assembler for the 8080 processor. It is used to convert assembly language source programs into machine code.

DDT

The DDT command (dynamic debugging tool) is used to test machine language programs.

How does this discussion of CP/M relate to the Atari ST?

The CP/M operating system is the most wide-spread operating system for eight-bit computers. Because it is a proven and solid system, it has been adapted to modern 16-bit computers as well. The first adaptation was CP/M 86, for the 8086 and 8088 processors. But from Atari's standpoint, the adaptation for the 68000 processor was more important. This version is called CP/M 68K.

Atari is using an operating system for the ST which offers the functions of CP/M 68K. In order to take the most advantage of the ST's features, BIOS functions were added to CP/M 68K to support all of then new hardware capabilities. This expanded CP/M 68K has the name **TOS** (Tramiel Operating System).

TOS operates very similarly to CP/M 2.2 for the eight-bit computers. It is also divided into the three components: the BIOS, BDOS and CCP. But there are a few major differences:

* The maximum memory size for CP/M 2.2 is 64K but for TOS the limit is 16MB.

* For CP/M 2.2, the operating system is loaded into memory from diskette when the computer is turned on. On the Atari ST, however, the TOS is stored in ROM. This has several advantages. It is available immediately after the computer is turned on since it does not have to be loaded from disk first. Second and more importantly, the user loses no memory space to the operating system.

* For CP/M 2.2, the operating system is always placed at the end of memory. The CP/M 68K operating system may be placed anywhere in memory.

CP/M 68K recognizes the same transient commands as CP/M 2.2. The DIR command has been expanded.

DIR

This command displays only *non-system* files.

DIRS

This command displays *system files* on the diskette. The operating system is no longer stored on reserved tracks of the disk but uses files of type .SYS.

SUBMIT

The SUBMIT command permits *batching* of commands. By placing a sequence of commands in a file, you can perform them in series as if they were entered from a keyboard.

3.2 BDOS FUNCTIONS

The following are the BDOS and BIOS functions available under CP/M 68K.

A user program can communicate with the peripheral devices via BDOS calls. Logical input and output devices are defined in the BDOS which are first assigned to physical devices in the BIOS. The following input devices are supported by the BDOS:

CON: STANDARD INPUT, USUALLY KEYBOARD
AXI: AUXILIARY INPUT, USUALLY SERIAL INTERFACE

Three output devices are available:

CON: STANDARD OUTPUT, USUALLY SCREEN
LST: LIST DEVICE, USUALLY PRINTER
AXO: AUXILIARY OUTPUT, USUALLY SERIAL INTERFACE

If a service is required of the BDOS by a user program, the BDOS is accessed with a TRAP instruction. The TRAP instruction causes the 68000 to execute an exception handling routine. When the 68000 encounters this instruction, it branches to a routine for handling this condition. Since the BDOS has many functions available, the number of the BDOS function is passed in register D0 of the 68000. Based on this number, a branch is made to the routine which performs the desired function. The additional registers of the 68000 are used to pass parameters for the functions. If a character is to be output to the screen, for instance, it is passed in register D1. On the other hand, if a character is to be read from the keyboard, a call to this BDOS returns the character in register D0.

Here's an example. To print the character **X** on the screen, a call to the appropriate BDOS function (number 2) might look like this:

```
002000 123C0058      MOVE.B #'X',D1 ; CHARACTER
002004 303C0002      MOVE.W #2,D0   ; BDOS #
002008 4E42          TRAP   #2      ; CALL
```

An area called the I/O-byte is defined in the BIOS. It's purpose is to assign logical devices to the physical devices. The I/O-byte is divided into four sets of two bits each. Each of these four groups is associated with a logical device.

I/O-Byte

	dev a		dev b		dev c		dev d	
bit	0	1	2	3	4	5	6	7

Four physical devices can be assigned with these two bits. The bit pattern "00" selects the standard device while a different bit pattern selects an alternate device. This makes it possible to redirect output from the screen to the printer, for example. In the same manner it's possible to reroute input from the serial interface, perhaps through a modem, instead of the keyboard.

In order to give you an overview of what functions can be executed by the BDOS, we will list them individually. In each case the function number is placed in register D0 before calling BDOS.

NUMBER FUNCTION

0 **SYSTEM RESET**

This function is used to return control back to CP/M 68K. This is normally at the end of a machine language program.

1 **CONSOLE INPUT**

This function inputs a character from the console. Control characters such as CR (carriage return) or LF (line feed) are also returned. This function waits until a character is entered at the console. The device from which the character actually comes is determined by the I/Obyte for CON: in the BIOS.

2 **CONSOLE OUTPUT**

This function outputs a character to the console. The ASCII code for the character to be output is placed in register D1. The device to which the character is sent is determined by the I/Obyte for CON: in the BIOS.

3 **SERIAL INPUT**

This function is like function 1 except that the character is input from the serial channel. This function also waits until a character is received. The device from which the character actually comes is determined by the I/O-byte for AXI: in the BIOS.

4 **SERIAL OUTPUT**

This function is similar to function 2 except that the character is output to the serial port. The ASCII code for the character to be output is placed in register D1. The device to which the character is sent is determined by the I/O-byte for AXO: in the BIOS.

5 **LIST OUTPUT**

This function outputs a character to the logical device LST:. The ASCII code for the character is placed in register D1. This device is normally assigned to the Centronics parallel interface, to which a printer is connected.

6 **DIRECT CONSOLE INPUT/OUTPUT**

This function is intended for special purposes where the normal BDOS functions do not work. Using this function, output to the screen cannot be stoppd with CTRL-S and cannot be redirected to the printer with CTRL-P.

7 **GET I/O-BYTE**

This function is used to examine the current contents of the I/O-byte.

8 **SET I/O-BYTE**

This fuction is used to change the current contents of the I/O-byte.

9 OUTPUT STRING

This function is used to output a string of characters. The string may be stored anywhere in memory. Register D1 contains the address of this string. The end of the string is terminated with a $ (dollar sign).

10 READ CONSOLE BUFFER

This function reads an entire input line from device CON:. The BDOS waits until the input is terminated with a RETURN. This function returns the number of characters entered and the memory address at which the characters are stored.

11 GET CONSOLE STATUS

This function informs the caller if a character has been input from the console. If a character has been entered, ff (Hex) is returned, otherwise zero is returned.

12 GET OPERATING SYSTEM VERSION NUMBER

This function returns the version number of the operating system. This can be examined from within a program.

13 RESET DISK

This function resets all disk functions and selects drive A.

14 SELECT DISK

This function selects a drive. The drive number is passed to the BDOS in register D1.

15 **OPEN FILE**

This function opens a disk file. The address of the file's FCB (file control block) is passed to the BDOS in register D1.

16 **CLOSE FILE**

This function closes a disk file. The address of the FCB is passed to the BDOS in register D1.

17 **FIND FIRST MATCHING ENTRY IN DIRECTORY**

This function searches the directory for the first matching entry. The address of the FCB is passed to the BDOS in register D1.

18 **FIND NEXT MATCHING ENTRY IN DIRECTORY**

This function searches the directory for the next matching entry. This is used if a wildcard is used in a file name. The address of the FCB is passed to the BDOS in register D1.

19 **DELETE FILE**

This function deletes a file from the directory. The address of the FCB is passed to the BDOS in register D1.

20 **READ SEQUENTIAL**

This function reads the next record (128 bytes) of an open file into a buffer. The address of the FCB is passed to the BDOS in register D1.

21 **WRITE SEQUENTIAL**

This function writes the current contents of a buffer to an open file. The address of the FCB is passed to the BDOS in register D1.

22 **MAKE FILE (WITHOUT OVERWRITE)**

This function creates a new file. It is similar to function 15, except no check is made to see if a file with the same name already exists on the disk.

23 **RENAME FILE**

This function renames a file. The address of the FCB which contains the old and new names is passed to the BDOS in register D1.

24 **RETURN LOGIN VECTOR**

This function returns the information about the currently selected drive.

25 **RETURN CURRENT DISK**

This function returns the current default drive.

26 **SET DMA ADDRESS**

This function sets the address of the buffer in which the data from the disk will be written.

27 UNUSED

This function is not used by CP/M 68K.

28 WRITE PROTECT DISK

This function protects a disk by prohibiting further writing to it.

29 GET R/O VECTOR

This function returns information about drives designated as "Read Only."

30 SET FILE ATTRIBUTES

This function is used to change file attributes. For example, you can designate a file as "read only".

31 GET ADDRESS DISK PARAMETER

This function returns the address of the disk parameter block. It contains information about the drive such as the number of tracks, number of sectors per track and the disk capacity.

32 GET/SET USER CODE

This function examines and changes the user number of a file.

33 READ RANDOM

This function allows you to read a specific block on a file.

34 WRITE RANDOM

This function allows you to write a specific block to a file.

35 **COMPUTE FILE SIZE**

This function returns the number of data records contained in a file.

36 **SET RANDOM RECORD**

This function passes the record number to be read or written with functions 33 and 34.

37 **RESET DRIVE**

This function resets the disk drive.

40 **WRITE RANDOM WITH ZERO FILL**

This function is similar to function 34, but the contents of the data record are set to zero values.

The following functions are found only in CP/M 68K.

46 **GET FREE DISK SPACE**

This function calculates the amount of free space available on the disk.

47 **CHAIN TO PROGRAM**

This function loads and executes the next program.

48 **FLUSH BUFFERS**

This function writes the file buffers to disk.

50 DIRECT BIOS CALL

This function enables you to directly call the BIOS from the BDOS.

59 PROGRAM LOAD

This function loads a program into memory without executing it.

61 SET EXCEPTION VECTORS

This function sets the 68000 exception handling vectors to the users routines.

62 SET SUPERVISOR STATE

This function actives the supervisor mode of the 68000.

63 GET/SET TPA LIMITS

This function examines or changes the starting address of user memory (TPA, transient program area).

3.3 THE BIOS OF THE ATARI ST

The BIOS for the Atari ST has additional functions which support the special hardware attributes of this machine. The standard BIOS calls have numbers starting from zero. The extended functions have numbers starting from 128.

Normally user programs do not use BIOS calls. Instead they use the logical functions provided by the BDOS.

The following listing contains a brief description of the BIOS calls.

NUMBER	FUNCTION
-1	**INITIALIZATION** The computer is completely reset by a call to this function.
0	**WARM START** The warm start function of the BIOS is called when a program returns control to the operating system.
1	**CONTROL STATUS** This function has the same task as the corresponding BDOS call. A value of ff (Hex) is returned in register D0 if a character is ready, otherwise a zero is returned.

2 CONSOLE INPUT

A character input from the console device is returned in register D0.

3 CONSOLE OUTPUT

The ASCII code contained in register D1 is sent to the console.

4 PRINTER OUTPUT

The character in register D1 is sent to the printer.

5 SERIAL OUTPUT

The character in register D1 is output to the serial interface.

6 SERIAL INPUT

A character is input from the serial port and returned in register D0.

7 FIND TRACK ZERO

The read/write head of the selected drive is placed over track zero.

8 SELECT DISK

The drive number to be selected is passed to this function which returns the address of the disk parameter header.

9 **SET TRACK NUMBER**

The track number is selected by the contents of register D1.

10 **SET SECTOR NUMBER**

The sector number is selected by the contents of register D1.

11 **SET DMA ADDRESS**

The address of the DMA buffer for disk data transfer is passed in register D1.

12 **READ SECTOR**

The sector set by previous BISO functions 9 and 10 is read from the disk. The contents of register D1 contains zero if the sector was read without error and -1 if an error was encountered.

13 **WRITE SECTOR**

The sector set by previous functions is written to the disk. The contents of register D1 must be set to 0, 1 or 2 depending on whether a write, a write to a directory sector or a write to a previously unwritten sector is to be performed. Following the call, register D1 contains an error code.

14 PRINTER STATUS

This function indicates if the printer is ready to accept the next character.

15 SECTOR CONVERSION

This function converts logical sector numbers to physical sectors.

16 UNUSED

This function is unused in CP/M 68K.

17 GET MEMORY RANGE

This function returns the starting address and length of the area available to the user.

18 GET I/O-BYTE

This function examines the I/O-byte.

19 SET I/O-BYTE

This function changes the assignments of the I/O-byte.

20 EMPTY BUFFERS

This function writes the contents of the disk buffers to the disk.

21 **SET ADDRESS FOR EXCEPTION HANDLING**

This function sets up the 68000 vectors for exception handling.

On the next pages you find the extended BIOS functions, denoted by function numbers over 127. They are used for handling the special features of the Atari ST hardware.

NUMBER FUNCTION

128 **CONFIGURE SERIAL INTERFACE**

This function sets the baud rate, handshaking method, number of bits per character, start and stop bits, and parity protocol of the RS- 232 interface.

129 **BUFFER FOR SERIAL TRANSMISSION**

This functions sets the size and address of the buffer for the RS-232 interface. If -1 is passed instead of new values, the current parameters are returned.

130 **RS-232 STATUS**

This function is used to determine if a character is available on the serial port, similar to the console status function.

131 **MIDI IN**

This function gets a character from the MIDI interface and returns it in register D0.

132 **MIDI OUT**

This function outputs the character contained in register D1 to the MIDI interface.

133 **MIDI STATUS**

This function is used to determine if a character has arrived over the MIDI interface.

134 **MIDI BUFFER**

This function sets the size and address of the buffer for the MIDI interface.

135 **KEYBOARD OUTPUT**

This function sends a character to the single-chip processor which controls the keyboard.

136 **MOUSE**

This function is the interface between GEM and the mouse.

137 **TIMER**

This function is the timer interrupt function for GEM.

138 **HARDCOPY**

This function outputs the contents of the screen to a graphics printer.Colour hardcopies can be created with a colour matrix printer.

139 **JOYSTICK**

The status of joysticks 0 and 1 is returned in registers D0 and D1.

140 **GET TIME**

This function returns the current clock time.

141 **SET TIME**

This function is used to set the clock time.

142 **SET COLOUR**

This function assigns an actual colour to a colour number.

143 **SET COLOUR PALETTE**

This function resets thecolourpalette which consists of a maximum of 16 simultaneously displayable colours.

144 **GET SCREEN ADDRESS**

This function returns the current address of the screen memory and the current display mode.

145 **SET SCREEN ADDRESS**

This function changes the display screen parameters.

146 **SOUND**

This function is used to program the PSG for sound output.

147 **INTIALIZE DISKETTE**

This function positions the read/write head at the starting position and the assigned buffer is cleared.

148 **READ SECTOR**

This function reads a physical sector. A physical sector is 1K large on the Atari drives.

149 **WRITE SECTOR**

This function writes a physical sector to the disk.

150 **READ TRACK**

This function reads an entire track from the disk to a buffer of the appropriate size.

151 **WRITE TRACK**

This function writes an entire track from a buffer to the disk.

152 **FORMAT DISKETTE**

This function formats a diskette.You must specify the number of tracks, the number of sectors per track, and the number of sides of the disk. This function also allows you to create a RAM disk.

4. TOWARDS A USER-FRIENDLY COMPUTER

Over the years, a large number of operating systems have been developed for computers. In general, the newer systems have included many new features and improvements over their predecessors.

In summary the major tasks of an operating system are:

* handling the commands entered by the user;
* loading user programs into the appropriate area of memory in response to the commands;
* giving these user programs access to the computer (execution);
* coordinating several user programs in a multi- user environment;
* managing the memory that is allocated for working storage;
* controlling and monitoring the flow of data among the different peripheral devices;
* resetting the normal state of the computer after the user program ends.

Most of these tasks are performed without the user even noticing that they are taking place. However, one area which has been particularly confusing to many users is the management of the peripheral devices.

Here the user is confronted by the rigid rules and procedures of the operating system. The procedures do work, but often presume that the user has in depth knowledge of the operating system. The user must be familiar with terms such as:

- **formatting**
- **file**
- **directory**
- **erase file**
- **backup or copy**
- **stat**

Terms such as these often frighten or confuse the novice computer user.

To make the computer easier and more friendly for the user, operating system designers have tried to simplify the way in which the user enters commands into the computer. The goals of these approaches are to make it possible for the user to perform productive work without extensive training or reference to volumes of technical handbooks.

Drawing from years of in depth study at Xerox Corporation's PARC, researchers observed that office employees do not want to give up their typewriters, filing cabinets and erasers in exchange for new technology. Instead they want to continue working in a familiar environment. So one approach to automation is to simulate this environment and its tools with the computer.

4.1 THE TRADITIONAL OFFICE

Some analyses found that the teacher in his study, the secretary in the reception room, the journalist in his open office and the business executive all work with similar equipment.

If we ignore the telephone, the list of equipment looks similar to this:

- desk
- typewriter
- blank paper
- correspondence
- file folders and portfolios
- several pens for notes and drawings
- an eraser for corrections
- a waste bin for hopeless cases

and more:

- a calculator for basic arithmetic operations
- a clock so that he/she knows when lunch break starts or as a reminder of other important appointments
- a copy machine so that important papers are not accidentally lost and forgotten.

Let's put ourselves at the desk of an office manager. A typical workday begins with the employee sitting down at his clean desk. His first assignment is to produce a sales report for the past month. He turns to his typewriter, inserts a piece of paper and types a few introductory lines. Next he looks through the sales folder containing last months records and spreads the daily sales logs out over his desk in order to get a better look at them. Next he types the daily sales totals. Then he reaches for his adding machine to calculate the monthly total. Everything seems to be going all right until he discovers that he overlooked several days sales because one sheet of paper obscured another.

So he inserts another sheet of paper into his typewriter, retypes everything, but this time not forgetting to include the missing day's sales. Now he can throw the first sheet of paper into the trash can and send the new report off to the boss. One further thing: he needs to make a photocopy of the report for the accounting department.

So to make the transition from a traditional office to an automated office as easy as possible, the work process should look the same when the office worker uses the computer. In addition, the worker's interaction with the computer should be easy to learn and understand.

4.2 PREREQUISITES FOR A FRIENDLY COMPUTER

A "friendly computer" must be blessed with a certain set of technical capabilities.

Due to the rapid advances in the micro-electronics field, new intergrated circuits have been developed that have enabled manufacturers to produce very capable computers, which the everyday user can afford. Likewise, software technology has made great strides: software for microcomputers has become more reliable, affordable and creative.

This advanced technology has enabled ATARI to create the ST with a resolution of 640x400 points at such an affordable price. This high resolution is suitable for representing objects on the screen -- objects such as a file cabinet, sheets of paper or even a waste bin. These objects have come to be known as *icons.*

A software innovation known as *windowing* has started to be widely used. On a computer screen, windows simulate several sheets of "paper" -- one on top of another. The user can "flip" through these sheets and display a specific page by selecting the appropriate window.

To be able to easily select a specific work task, or to select from among the different windows, a hardware innovation called a *mouse* was chosen. It is the use of this mouse as an input device that the Atari hopes to revolutionize the home computer market. While it is true that computers such as the Apple Macintosh use these same techniques, Atari has been able to do the same at such an affordable price. No longer is this exciting computer technology the privilege of a few. Instead the Atari ST makes it possible for the masses to afford a computer that can be operated through graphic oriented displays.

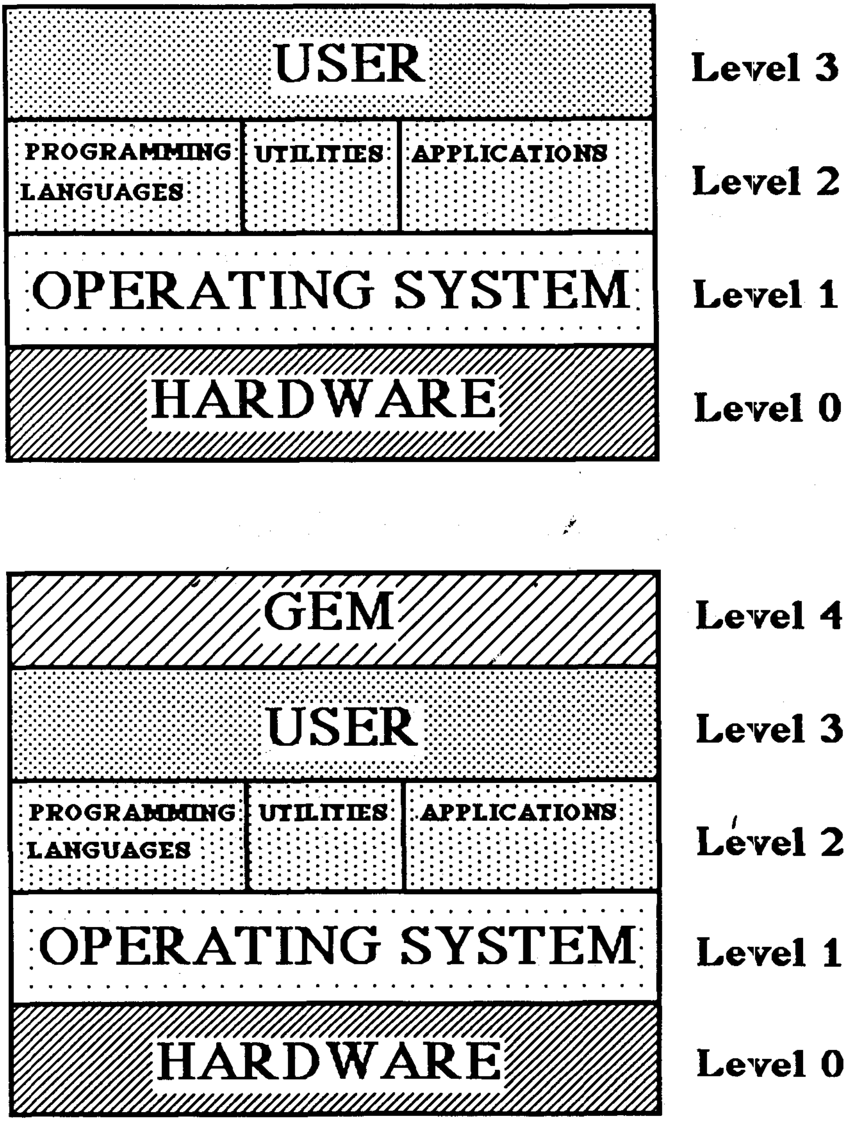
4.3 THE MOUSE

The mouse is a small input device,that houses a small ball whose rotation in two axes can be measured, either mechanically or optically. If you turn a mouse on its back, you can see that it's a variation of the more familiar trackball used in many video games. With a trackball, the user rotates the ball in the desired direction with short movements of the palm of the hand. With a mouse, the user moves it in a desired direction over a flat surface.

The operating principle of a mouse is based on a coordinate system where a movement in a given direction represents a given **x and y** value. If the flat surface over which the mouse is moved is said to correspond to the computer's display screen, then each point on the screen can be assigned an x-y coordinate pair. Thus a movement by the mouse represents a corresponding movement on the screen.

If software is written to support a mouse, then it's possible to replace the keyboard for many tasks. One method is to display a menu of choices on the screen. The mouse is used to position an arrow or cursor to the desired choice and a button on the mouse is pressed. This is called *clicking* the mouse. So you can select a function with two simple steps: moving the mouse and clicking it. You might even think of the mouse as an extension of your finger where you point to the symbols displayed on the screen. The Atari ST is designed to be used with a mouse.

FIG. 4-1 USER INTERFACE



4.3.1 WORKING WITH THE MOUSE

How is the mouse used to perform various tasks?

The ST's impressive hardware is not very useful without appropriate software. Let's see how we can do productive work with the ST.

Before using a blank diskette, it must be formatted. One of the functions of the CP/M 68K (or TOS if you prefer) operating system is formatting a diskette.

With most other computers, the user must know the name of the command (FORMAT) to perform a desired function. As is the case with many of these commands, one or more parameters must be entered as part of the command. So the user must also know how many parameters are required and the order and syntax of these parameters. Only then can he enter the full command at the keyboard.

One alternative to entering the command at the keyboard is to display a list of functions on the screen. Then using the mouse you can click to the desired function. The user needs only remember the correct command, but not a complicated syntax. Although simpler to perform than typing the command, the method is a long way from the automated office.

Now let's introduce a slightly different concept. In place of the list of functions, why not represent them as icons (graphic symbols) on the screen? The Atari ST does just this by using an operating system called

GEM. GEM (Graphics Environment Manager) is a graphics oriented operating system from Digital Research. In practice, GEM is not really a full operating system. Rather it is an interface between the user and the operating system. It's purpose is to simplify the communication between the user and the TOS operating system.

Figure 4-1 illustrates how the hardware and software components of the ST fit together.

At the lowest level is the hardware. It is not directly controllable by the user. Instead, the operating system at the next level controls the hardware. The user at level 3 communicates with the operating system through programming languages, utilities or application programs at level 2.

By introducing another level of communication we can reduce the amount of knowledge that the user requires in order to interact with the language, utilities or application programs. So move the user interface up to level 4 and place GEM at level 3. Now, the only way to use the ST is through the GEM interface.

5. WORKING WITH GEM

After turning on the ATARI ST, the screen displays a pictorial representation of a desktop. It is divided into the working surface of the user and the system menu lists.

Icons appear on the desktop along the right side of the screen. These icons are figures of the peripheral devices connected to the ST. For example, two light coloured diskettes represent the two ST floppy disk drives - one called A: and the other B:. At the lower right hand of the screen is an icon of a waste bin.

A small black arrow appears in the middle of the screen representing the position of the mouse. This is called the mouse pointer or arrow. The mouse is the main input device that the ST uses under GEM.

There are two fundamental operations that can be performed with a mouse: selection and dragging.

To select an object that appears on the screen, simply move the arrow across the screen with the mouse until it is positioned over the desired icon and then click the mouse button. The selected icon is then displayed in reverse on the screen, in black.

To move an object from one location to another on the screen, you must drag it. To drag an object, move the arrow across the screen with the mouse until it is positioned over the desired icon, press and continue to hold the mouse button, and then reposition the icon on the screen by moving the mouse. When the icon is in the desired position, release the mouse button. As long as you continue to press the button, the icon can be moved from place to place on the screen.

5.1 MENUS

As already mentioned, GEM uses both icons and menus. When the ST is first turned on the following menu is displayed at the top of the screen:

DESK FILE VIEW OPTIONS

Now if you position the arrow over one of the menu choices, press and continue to hold the mouse button, a *pull-down menu* will appear. A pull-down menu works like a common household window-blind. The blind may be pulled down when needed or retracted if not needed. If needed, a pull-down menu displays a further set of choices.

So by positioning the arrow over the DESK menu selection and pressing the mouse button, the following pull-down menu appears and display a set of functions that you may use on your automated desktop:

DESK FILE VIEW OPTIONS

DESK TOP INFO
BREAKOUT
CALCULATOR

The pulldown menu for FILE contains all the functions for manipulating files, whether the file is a program, document, or the data file:

DESK FILE VIEW OPTION

OPEN
SHOW INFO
NEW FOLDER
CLOSE FOLDER
CLOSE WINDOW
FORMAT
TO OUTPUT
QUIT

The pull-down menu for VIEW displays the contents of ST's disks. You can display the files in a variety of ways. The contents may be displayed as icons or as lists. In addition, they may be arranged by name, date, size or type. The pull-down menu looks like this:

DESK FILE VIEW OPTIONS

SHOW AS ICON
SHOW AS TEXT
SHOW BY NAME
SHOW BY DATE
SHOW BY SIZE
SHOW BY TYPE

The pull-down menu for OPTION allows you to set system-specific parameters:

DESK FILE VIEW OPTIONS

INSTALL DISK DRIVE
INSTALL APPLICATION
SET PREFERENCES

The install functions are used to adapt peripheral devices or other programs to GEM. Other miscellaneous system parameters can be changed with SET PREFERENCES.

GEM is designed to be as user-friendly and forgiving as possible. Therefore, confirmation is always requested before a copy or deletion is performed. For example, when deleting a file a *dialog box* appears on the screen that looks similar to this:

CONFIRM DELETE: YES
NO

At his point, the user can cancel the operation by clicking NO on the dialog box.

By using these four pull-down menus and the icons, you can perform all of the usual operations such as formatting diskettes, deleting or copying files, viewing the directory of a disk, and more, without ever once having to use the keyboard.

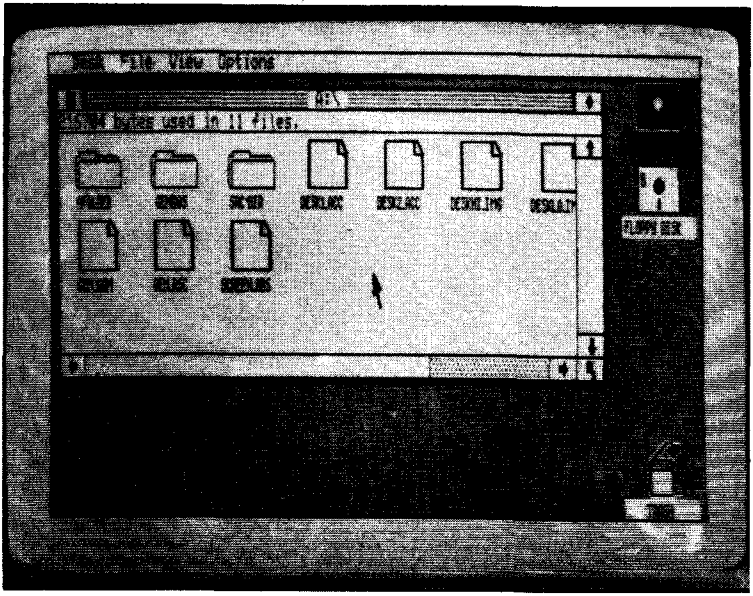
Here's an example:

Select a disk drive by positioning the arrow over the disk icon and pressing the mouse button. The icon now appears in reverse.

Suppose you want to know which files are on the disk in this drive. Position the arrow to **FILE** and press the mouse button without releasing it. A pull-down menu appears. Drag the arrow to the **OPEN** choice and release the button.

Immediately, the pull-down menu disappears and a new window appears (the window is said to have opened) displaying the contents of the diskette on the screen as icons.

To show the contents as text instead of icons, position the arrow to **VIEW** and press the mouse button without releasing it. Another pull-down menu appears. Drag the arrow to **SHOW AS TEXT** and release the button. Again, the pull-down menu disappears and the contents of the window is rearranged. The disk files are now displayed as text instead of icons.

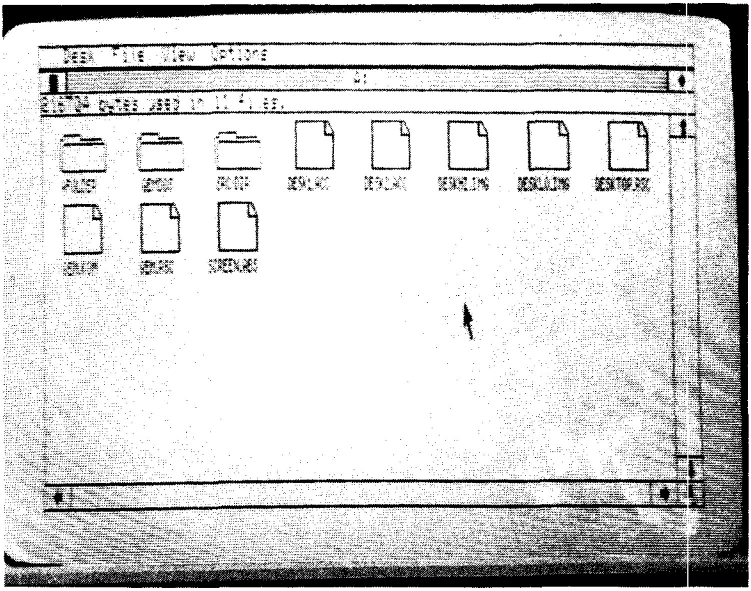


Now let's take a closer look at the GEM windows.

A GEM window is made up of several elements: a border surrounds a light background; arrows are in the corners of the windows; a black rectange is found in the upper left and lower right corners.

The arrows are used to *scroll* the screen. If when displaying information on the screen, it will not all fit within the window dimensions, you can use the arrows to scroll the information within the window. To scroll downwards, for example, you would position the mouse arrow over the down scroll arrow located along the right-hand edge of the window and click the mouse button. The information on the screen will then scroll downwards.

To display more information on the screen, you can also change the size of the window. Do this by dragging the window's size-box located in the lower right-hand corner of the window to a new location. As you drag the size-box, you can see how the dimensions of the window change. The maximum size of a window is determined only by the size of the work surface available. The window may be enlarged to cover even the TRASH icon for example.



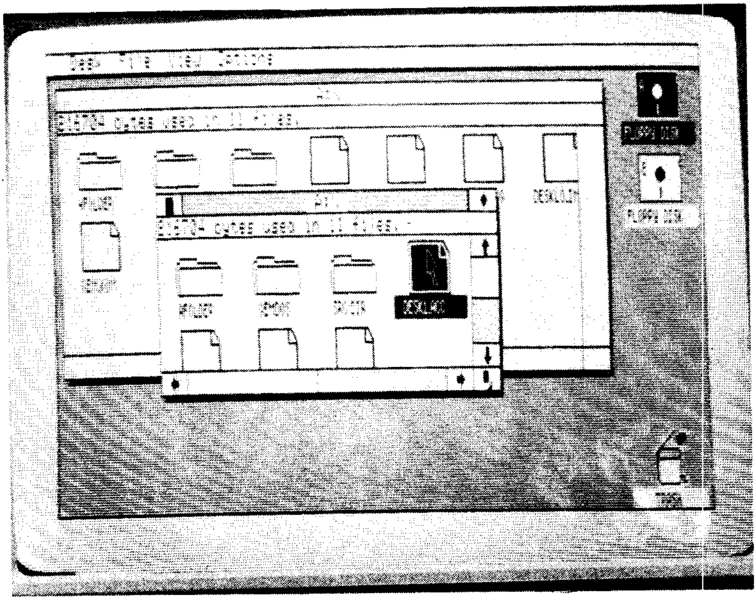
As you can see, the menu titles remain displayed at the top of the screen.

If you have completed using a window, you should **CLOSE** it. Do this by clicking the close box in the upper left-hand corner of the window. The window disappears and any information that was hidden by the window is again visible.

While all of this is taking place on the screen, the processor is performing many tasks. In order to restore a screen when a window is closed requires that the previous contents of the screen be saved. If multiple windows are superimposed on the screen, they too must be saved. To do this the operating system must have a lot of memory available to use and must work quickly since the user does not want to wait long for the new screens.

In the picture on the next page, you can see that a section of the border is cross-hatched. This means that only part of the available information can be displayed within the window. To display the rest of the information, you can scroll the window using the scroll arrows.

You'll also notice that the icon for a file called **DESK1.ACC**. It appears in reverse, meaning that it is selected.



Instead of selecting an icon and then going to the **FILE** menu to **OPEN** a file, you can position the mouse arrow to an icon and click the mouse twice in quick succession (called *double clicking*). Double clicking is a shortcut to opening a file. By double clicking the icon for **DESK1.ACC**, we can begin executing that program.

5.2 WINDOWS UNDER GEM

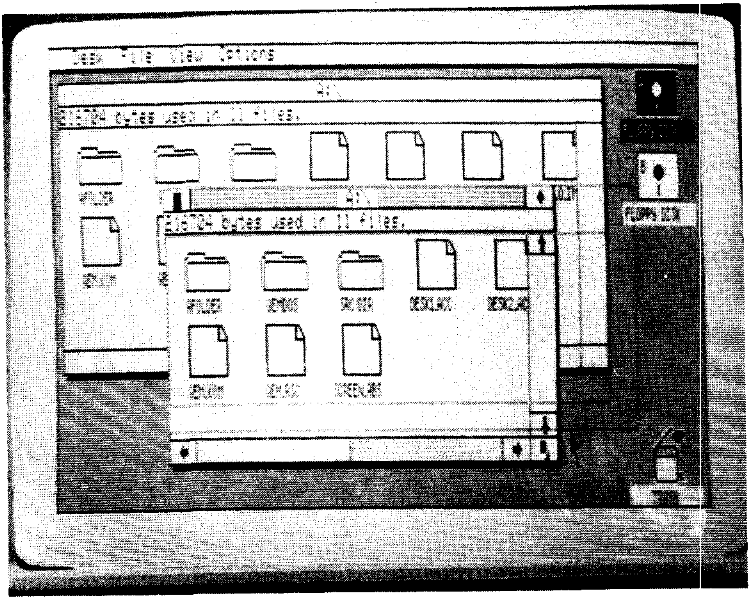
A window consists of a border surrounding the viewing field. At the top of the window is the title box displaying the name of the file or disk. Beneath the title box is a box displaying the amount of storage space that the files contain. It looks like this:

216704 bytes used in 11 files

A window currently in use contains a close box. A window that is overlayed by another does not have a close box.

On occasion you might want to enlarge a window to make as many objects as possible visible at one time. On the other hand, you might want to reduce the window size in order to see a partially covered window underneath. To change the size of the window opening, drag the size box until the window is of the desired dimensions and then release the mouse button.

You can also relocate an entire window on the screen. To do this, drag the title line of the window to the new location and then release the mouse button. You'll note that as the window is being dragged to a new location, only an outline of the window is moved. The window itself is not relocated until the mouse button is released. See the picture on the next page.



At any given time, only one window may be *active*. To make a window active, you simply move the mouse arrow to the window's border and press the button. GEM moves this window on top of any other papers in the stack. When you complete your work on this window, you can active another window in the same way: clicking the new window's border.

Fig. 5-1 GEM WINDOW LAYOUT

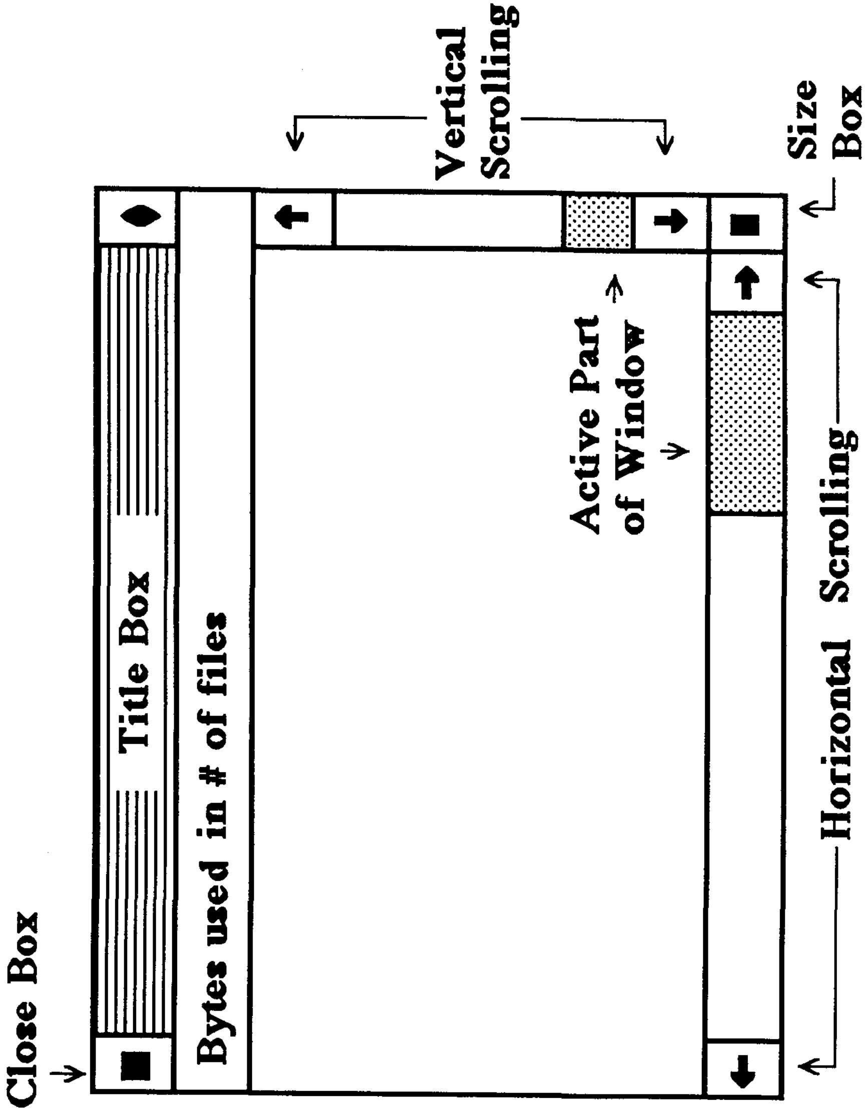
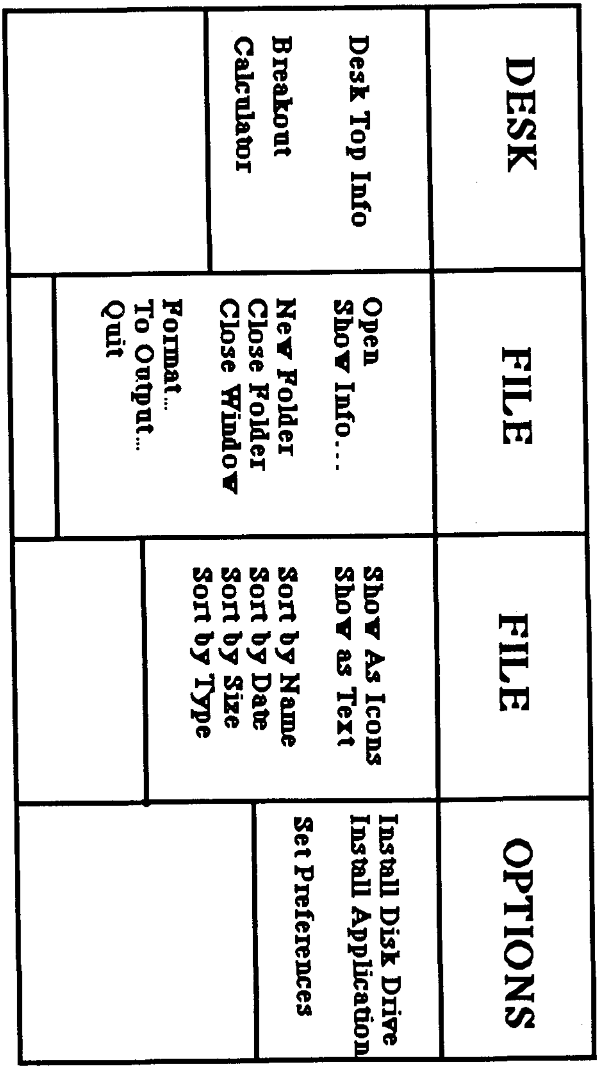
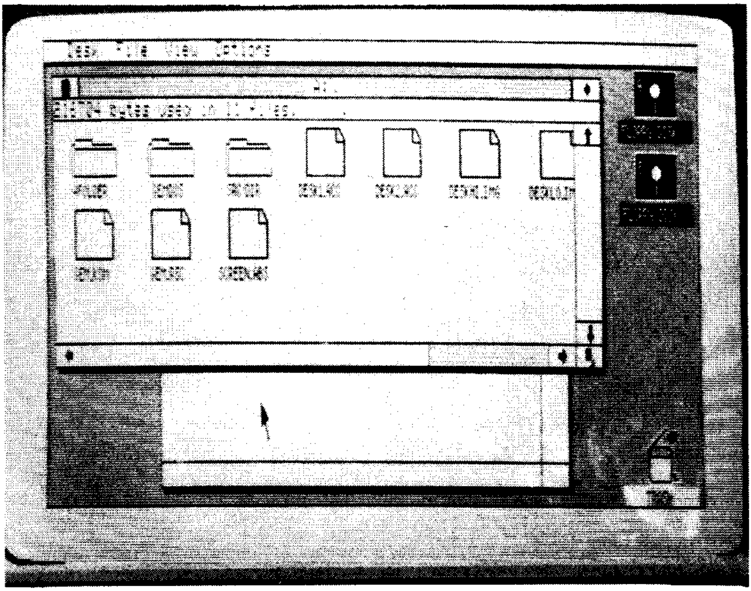


Fig. 5-2 GEM MENU DIAGRAM



5.3 THE DISK DIRECTORY

You can display the directory of a diskette by using the **VIEW** option on the desk. By default, the directory is displayed as icons as in the following picture.



The amount of disk storage that is occupied is displayed in addition to the file names and types.

Among the different file types are:

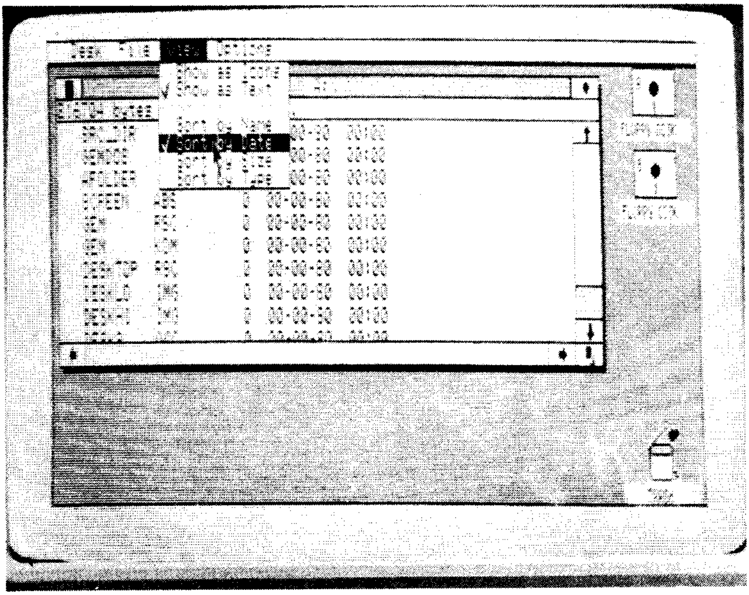
- * folders
- * programs
- * data

The icon for program or data files is a sheet of paper.

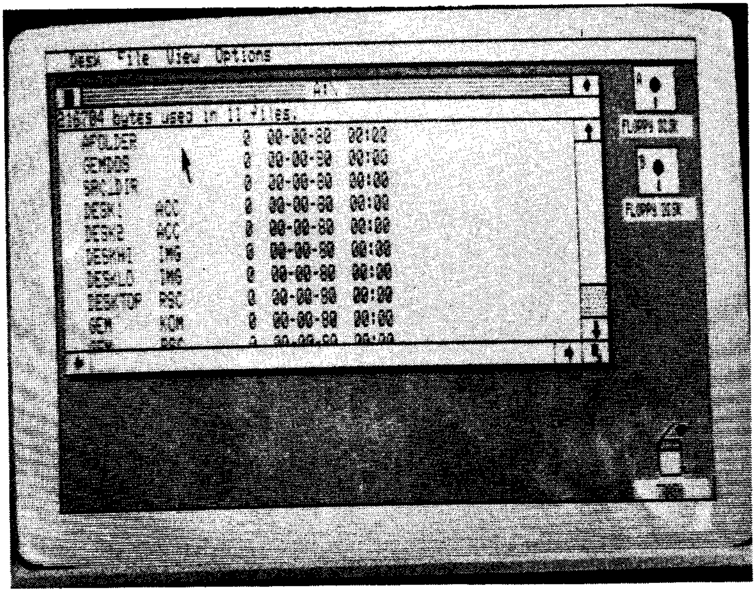
The icon for a folder is a symbol that looks like a familar manila file folder. The folder icon is a way to group an arbitrary number of program or data files. By grouping these files in a folder, the user can more easily organize the information on the disk. To see the contents of a folder, you merely open the folder (by either double clicking or by selecting and then OPENing it).

As an alternative to displaying the directory as icons, you may also display the contents alphabetically by file name; alphabetically by file type; chronologically by creation date; in descending order by file size.

You can select individual files by using the mouse arrow. In addition to information about the individual files or the folders, you can also call up data on the status of a diskette. The operation SHOW INFO in the pull-down FILE menu is used for this purpose.



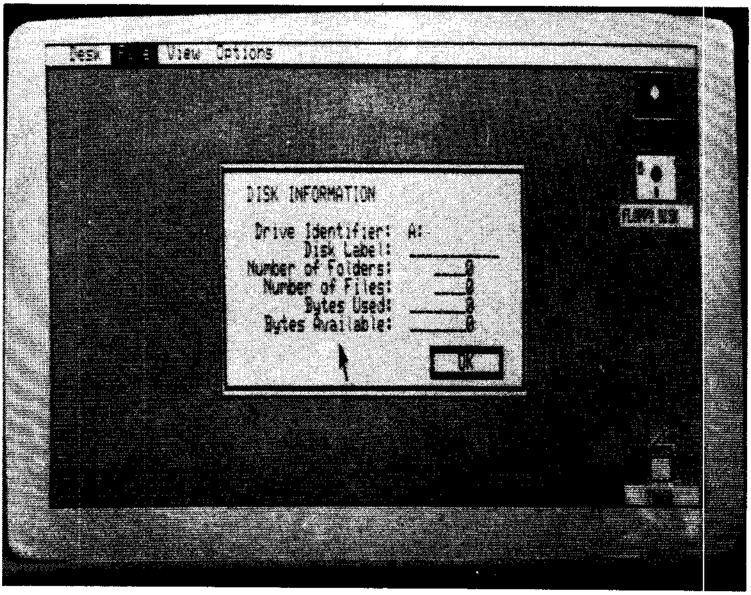
```
SMALLDIR              00-00-80  00:00
GEMDOS                00-00-80  00:00
AFOLDER               00-00-80  00:00
         ABS       0  00-00-80  00:00
GEM      PSC       0  00-00-80  00:00
GEM      KOM       0  00-00-80  00:00
DESKTOP  PSC       0  00-00-80  00:00
DESKLO   IMG       0  00-00-80  00:00
DESKHI   IMG       0  00-00-80  00:00
```



```
216704 bytes used in 11 files.
AFOLDER            0  00-00-80  00:00
GEMDOS             0  00-00-80  00:00
SMALLDIR           0  00-00-80  00:00
DESK1    ACC       0  00-00-80  00:00
DESK2    ACC       0  00-00-80  00:00
DESKHI   IMG       0  00-00-80  00:00
DESKLO   IMG       0  00-00-80  00:00
DESKTOP  PSC       0  00-00-80  00:00
GEM      KOM       0  00-00-80  00:00
```

In this case, drive A: contains no data at all. You should notice that the number of files and folders, the amount of unused space on the disk and the name of the diskette are displayed.

After viewing this information, click the **OK** field in the dialog box to return to the previous window.



DISK INFORMATION

Drive Identifier: A:
Disk Label: ____________
Number of Folders: 0
Number of Files: 0
Bytes Used: 0
Bytes Available: 0

5.4 WORKING WITH FILES UNDER GEM

Backing up files is one of the most fundamental file operations. Using GEM, it is also one of the easiest operations to perform.

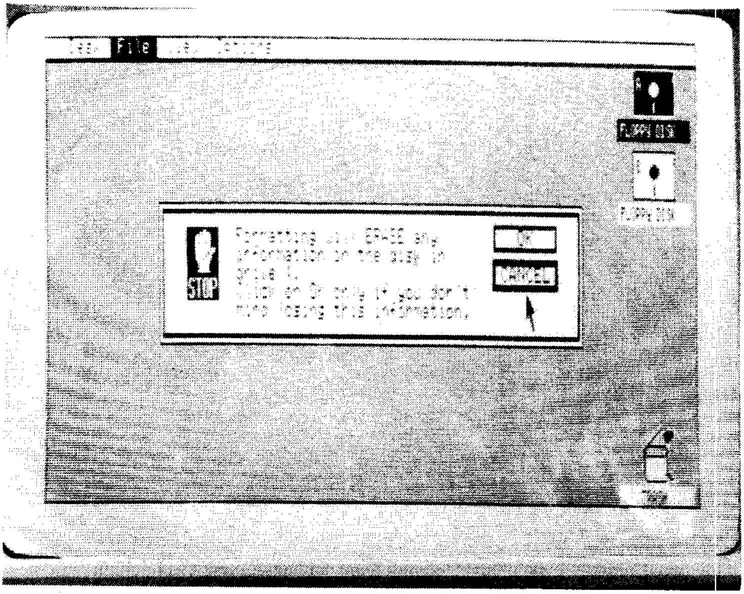
Let's see how we would back up a file using the Atari ST. The source files are on drive A: and we will copy them onto a disk in drive B:.

The first step is to select drive B:. Do this by clicking the icon labeled **FLOPPY DISK B:**. If the diskette in this drive is blank, format it by selecting clicking **FILE** from the menu and dragging the arrow until the **FORMAT** operation is selected by releasing the mouse button. Since GEM is user-friendly you'll see the following message on the screen:

STOP
Formatting will ERASE any
information on the disk in
drive B:.
Click on OK only if you don't
mind losing this information.

If you've made a mistake, you can leave this procedure by moving the mouse pointer to CANCEL. By clicking OK the formatting begins.

When the light on the drive goes out, we're ready for the next step.



Formatting will ERASE any
information on the disk in
drive A.
Click on OK only if you don't
mind losing this information.

To copy a file, simply drag the icon for the the desired file until it is positioned over the icon of the destination drive and releasing the mouse button. Copying begins immediately.

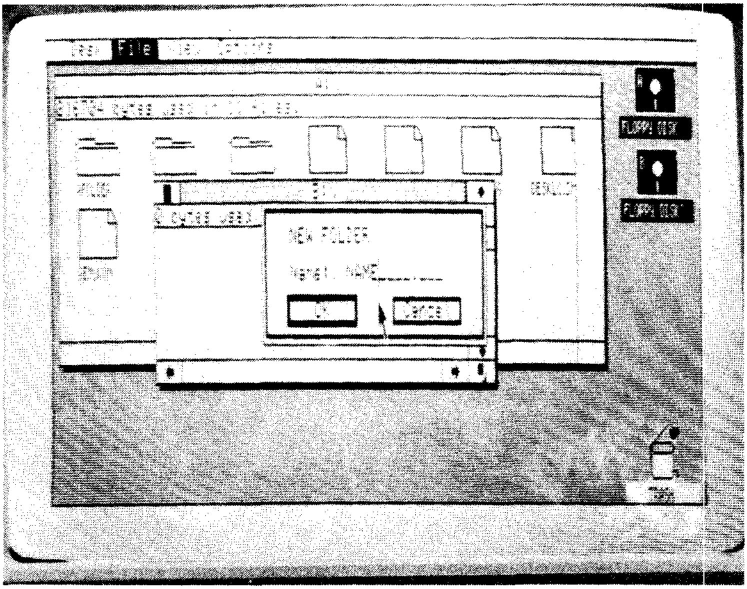
You can also manipulate multiple files as a group. To do this, the icons for the individual files must be displayed on the screen next to each other. To select the group, it's necessary to form an imaginary rectangle around the desired icons. First, position the mouse arrow to one of the two diagonals of the rectangle and press the button. Now drag the arrow to the opposite corner of the rectangle. As you are dragging the arrow, a dashed line appears on the screen to show you the icons which have been selected. When all of the desired icons have been selected (are within the dashed rectangle) release the mouse button. All selected icons will now appear in reverse.

By moving one of the selected icons, all of the icons appear to move together. The selected group of files are displayed as outlined icons as they are dragged. To copy these files to a backup disk, drag the icon of one of the group of files until it is positioned over the destination disk drive and release the button. The selected files are copied to the backup disk one at a time.

This method of manipulating files is very easy to learn and very powerful as well. It can prove dangerous too since it's possible to toss many files at a time into the **TRASH**.

So we've seen a very convenient way to group files together. Now let's look a another way. Go the the **FILE** option, drag the mouse arrow to **NEW FOLDER** and release the mouse button. A new window appears on the screen. Now you'll have to resort to the keyboard to enter a name for the file folder. A folder name consists of up to eight characters with an optional three character extension. When you have typed in a file folder name, click the **OK** box.

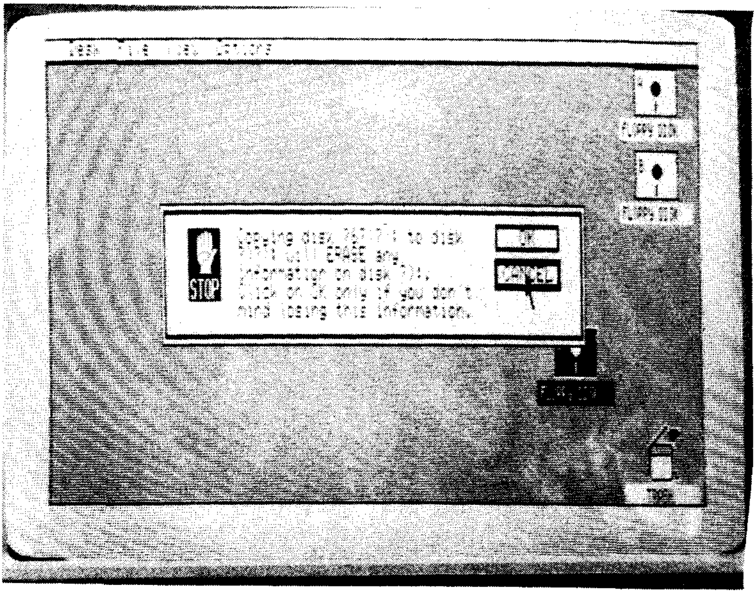
To place a file into this folder, drag its icon until it is positioned over an empty folder and then release the mouse button. The file icon immediately disappears from the screen. The file has been placed into the folder. Naturally, you can place as many documents to the file folder or create as many folder as necessary to achieve your desired degree of organization.



The final example, is the copying of entire diskettes. This is how you copy an entire diskette:

Use the mouse arrow to select the source disk drive, drag the icon until it positioned over the destination drive and release the mouse button. You'll see the following message appear on the screen:

Copying disk A: to disk **OK**
B: will ERASE any
information on disk B:. **CANCEL**
Click on OK only if you don't
mind losing this information.

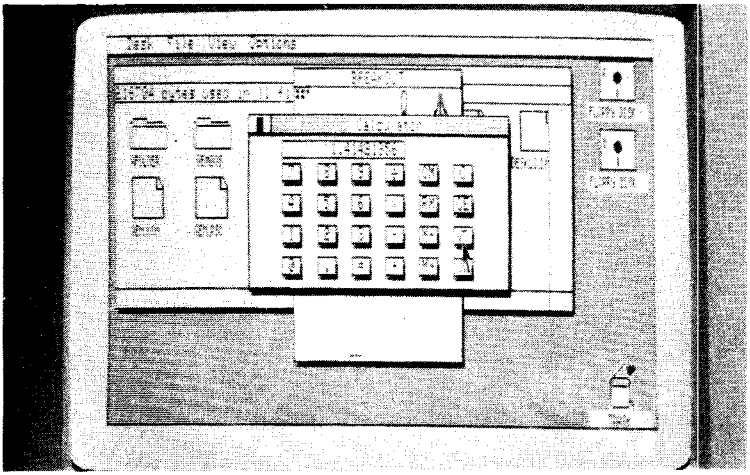


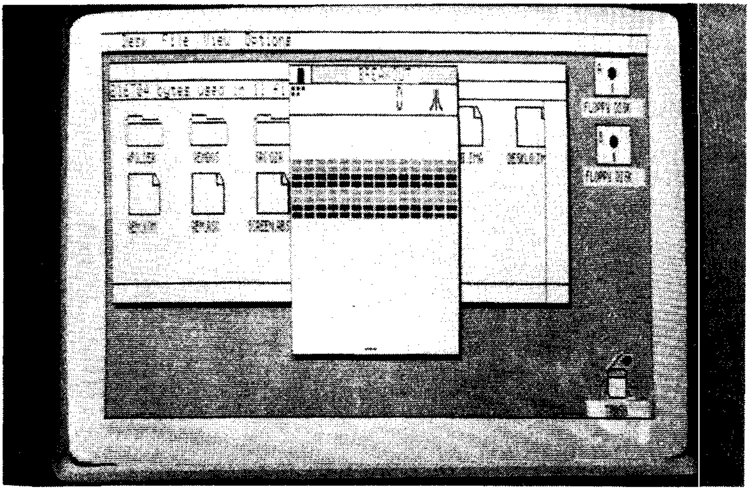
Copying disk [illegible] to disk
[illegible] will ERASE any
information on disk [illegible].
Click on OK only if you don't
mind losing this information.

Before leaving our discussion of GEM, we want to quickly point out a few other features.

When working at a desk, you occasionally need to use a pocket calculator for number work. By dragging the **DESK** option to **CALCULATOR**, the ST will reveal a built-in calculator. It has a standard calculator that includes percent, square root and several memories. So the Atari ST desktop is ready to serve even more of your needs.

When you find yourself bored at your Atari ST desktop, you can find diversion with the BREAKOUT arcade game. By dragging the **DESK** option to **BREAKOUT**, a game board will appear on the screen. By using the mouse, you can play the classic Atari in vivid color.





5.5 INSIDE GEM

GEM is an acronym for Graphics Environment Manager. The complete GEM software package is a graphics-oriented user interface designed to simplify the operation of the computer.

There are two major components to GEM. They are the VDI (Virtual Device Interface) and the AES (Application Environment Services).

The Atari ST was designed with this simplicity in mind. To make working with the ST as easy as possible, all user interaction with the operating system and application are meant to be handled in an identical fashion. This considerably shortens the training time to learn to use a new application, since the basic operations of using the mouse, icons, pull-down menus and windows remains the same. In addition, software designed for GEM works independent of the computer hardware configuration. So if the hardcopy device for the ST is changed from a dot matrix printer to a more expensive plotter, the output is easily produced.

This compatability is achieved through the *virtual device interface*. The idea behind this is a software *driver* which is responsible for the output of data to hardware periperal.

All input and output for the ST are sent to the VDI and not directly to the operating system. The VDI accepts incoming data and reproduces it on the screen, on the printer, on the plotter, etc.

Another component of GEM is AES or application environment services. The AES monitors the input devices such as reading the keyboard, evaluating the movement of the mouse, and checking the mouse button. The AES also controls output formatting such as windows, pop-down menus and icons.

These two parts of GEM, the VDI and the AES, along with the TOS form the block of system-specific software. Imbedded in this system is the DESKTOP which can call the various utility programs, the TOOLS. These might include an icon editor with which you could create your own icons.

5.6 VIRTUAL DEVICE INTERFACE

The VDI itself, consists of two parts: namely the GDOS, an operating system for controlling the graphics-oriented output devices, and the driver which contains the information concerning the character sets (or fonts) to be used.

This differentiation suggests one essential difference between GEM and earlier operating systems. On conventional computers, there are two types of screen displays: a *TEXT* mode and a *GRAPHICS* mode. In the TEXT mode display, one byte of the screen memory is interpreted as a character according to the assignments of the ASCII codes. In the GRAPHICS mode, on the other hand, when between 50,000 and 200,000 points are addressed, a single bit represents only one of these points.

As a result, the size of the screen memory is dependent on the screen display mode. Further, trying to combine text and graphics on the same display are made more difficult.

Using GEM, there are no provisions for a pure TEXT mode; the information is always displayed in the GRAPHIC mode.

To display an alphanumeric character at a specific location, the bits in the corresponding screen memory are set to the pattern of the desired character. This is called *bit-mapped* character representation. Since any pattern can be displayed, the user is free to use any set of alphanumeric patterns. Such a set of patterns is called a *character set*. Different character sets may be stored on diskette and used in place of a standard character set. A common use for user-defined character sets is to display foreign characters or to substitute a new *font*. Thus alphanumeric characters are handled just like graphic patterns in a bit-mapped system.

Here's another example of the drawback of a separate TEXT and GRAPHIC display mode. Suppose you want to display a square box and a perfect circle on the screen and then get a hardcopy of them on the computer's dot matrix printer. Using most computers, the square is reproduced as a rectangle and the circle as an ellipse. This is due to the differences in point representation on the screen and the printer.

The VDI can overcome this problem. The VDI recognizes two different coordinate systems, namely:

* **NDC** - Normalized Device Coordinates, and
* **RC** - Raster Coordinates.

Raster coordinates correspond exactly to the physical capabilites of a display device. For the Atari ST display screen, this is either 320 X 200 or 640 X 400 points.

Normalized device coordinates correspond to an idealized screen surface. Using normalized coordinates, a quadrilateral that appears square on the screen is made to look square when printed as hardcopy. For a coordinate system of say, 600 x 200, the height to width relationship between the coordinates on the screen is 1:1.8. That is, one screen pixel in the horizontal direction has the same dimensions as 1.8 pixels in the vertical direction.

With the VDI, a software *device driver* carries out the necessary conversions for NDC. The output appears in the same proportions on all peripheral devices.

To do this, GEM use *meta-files.* These files contain information required to exchange data among peripheral devices and individual GEM applications. In a meta-file, the physical size of the output device and the coordinate system for the data to be transmitted is defined.

5.7 APPLICATION ENVIRONMENT SERVICES

The AES is composed of several components. The most important are the dispatcher, the screen manager and the subroutine library.

The *dispatcher* is responsible for handling multiple applications that are able to run simultaneously. Since the 68000 processor is so fast compared to the eight-bit processors, it is possible for the ST to perform more than one task at a time. This is called *multitasking*. Later versions of GEM may support multitasking.

The subroutine library contains the appropriate routines for manipulating the windows, handling the mouse, and so on. The screen manager assumes control of the mouse once it is outside of a window.

6. COMMUNICATION BETWEEN MAN AND MACHINE

We've talked quite a bit about the Atari ST from both the hardware and software standpoint.

But many of us are interested in nitty-gritty techniques of using the ST.

How do you control the sophisticated chips in the ST? How do you use the fantastic sound synthesizer that is built-in?

How do you set and reset the individual pixels on the display screen?

Just as we communicate with each other using the English language, you can communicate with the ST using a variety of languages.

6.1. COMPUTER LANGUAGES

There are a wide variety of computer langauges. Hopefully, each different language serves a certain user need.

A computer language must first of all be unambiguous. Each individual language element must have an exact defined syntax and meaning; ambigous interpretations are not allowed.

To the casual user, the computer is an intelligent machine. But the fact is, even the fastest, most expensive computer is only a simple machine. Without the correct instructions, it is incapable of performing any useful work.

This short example shows that even a simple addition requires a program to tell the computer how to proceed:

1. Get the first number
2. Get a second number and add it to the first
3. Display the sum of these two numbers

Even though it's easier to perform the addition in your head or on a pocket calculator, here's a sample solution on a computer. We must know some facts before we proceed, for example, from where do we get the two numbers?

Two memory locations may be designated as the source of these numbers. The instruction sequence might look like this:

1. Load the first number into the accumulator from memory location 1.
2. Get the second number from memory location 2 and add it to the number in the accumulator.
3. Write the sum (the number in the accumulator) to memory location 3.

The 68000 processor could follow this algorithm, but not before we have made sure that the two numbers are actually in the appropriate memory locations. Furthermore, we must be able to examine the contents of location 3 to find the result of the calculation.

You might want to write this program in machine language or assembly language. Here you would use the instruction set of the processor which we introduced earlier.

If you take a look at an assembled program, you'll see a series of numbers in the printed listing. These numbers are the hexadecimal representation of the machine language. If you can imagine the hexadecimal numbers as a bit pattern containing only 0's and 1's, then you'll see the actual machine language program as the 68000 processor sees it.

ASSSEMBLED 68000 PROGRAM

```
    HEX CODE                         SOURCE CODE

1   00000000                                .text
2                                    * This program adds the first
3                                    * three integers and stores the
4                                    * result in memory
5   00000000  303900000000           start: move.w   a,d0   Load first number
6   00000006  D07900000002                  add.w    b,d0
7   0000000C  D07900000004                  add.w    c,d0
8   00000012  33C00000000A                  move.w   d0,d   Store answer
9   00000018  4E75                          rts             Return to CP/M
10  00000000                                .data
11  00000000  0001                   a:     .dc.w    1      Numbers to add
12  00000002  0002                   b:     .dc.w    2
13  00000004  0003                   c:     .dc.w    3
14  00000006  0000                   d:     .dc.w    0      Answer goes here
15  00000008                                .end
```

It is this series of 0's and 1's that the processor receives during execution. Supppose for a moment, that you had to communicate with the 68000 by using only these binary numbers. You would have to know the series of numbers for each different instruction and also the series of numbers for the location of the data which the processor required. Needless to say, programming a processor using binary machine code would be very error prone and time consuming.

Assembly language is a great step forward in programming. The bit patterns for each instruction are replaced by easy to remember *mnemonic* abbreviations. It's easier to remember a mnemonic such as LD (for **L**oa**D**) or LDA (for **L**oa**D** **A**ccumulator) instead of a cryptic bit pattern.

To summarize machine and assembly language programming:

1. The instructions perform relatively fundamental operations such as data manipulation and basic arithmetic. For this reason a program must be broken down into a large number of individual instructions. In doing so, the program becomes very long. Increasing the number of instructions also increases the liklihood of errors.

2. Machine or assembly language programs are rather difficult to read and change. Therefore program maintentance is slow and expensive.

3. The mnemonic instructions depends on the processor used and architecture of the computer system. Exchanging programs among different computers is therefore very costly or impractical.

4. The programmer must be very familiar with the operation of the computer and all peripheral devices. He must know how to access, the keyboard, screen, printer, disk drive, etc.

We don't want to downplay the use of assembly language programming. An assembly language program can take full advantage of the processor. Each of the instructions is available to the assembly language programmer where he may hand-tailor the software for highest execution speed. He has total control over the memory used for the program and data.

6.2 PROBLEM-ORIENTED PROGRAMMING LANGUAGES

As computers became more widely used, the base of people using them became wider as well. Not only were the physicist and mathematicians using the computer but others from other disciplines were using them as well.

The physicist and mathematician were fairly well-suited to learning computer programming. They found the transition from hand-written algorithms to computer algorithms rather straight-forward.

Others, however, found the transition much more difficult. Furthermore, it was too expensive to describe a problem to a programmer so he could write a program to solve it. For this reason, new artificial languages were developed so that the user could program a solution to his problem in a more familiar terminology.

The first major language to appear was called **FORTRAN**, for **FOR**mula **TRAN**slator. It was highly mathematical in nature and was developed mainly to suit the mathematical needs of scientists and engineers.

Many other languages followed, a few of which came to fame and wide-spread use, while others which ran on only a few individual computers lasted only a short time.

Languages such as COBOL, the **CO**mmon **B**usiness **O**riented **L**anguage, were written especially for use in business data processing.

These two languages were designed as *compiled langauges.* The compiler checked the language syntax and grammar for correctness and converted the individual statements into equivalent machine language instructions. Finally, this resulting program was executed.

The advantages of this procedure are obvious:

* The programmer can describe his problem in terms that he is familiar.

* The compiler can translate these terms into the machine language of the computer. In fact, a compiler can be written to translate the problem- oriented language into the machine language of any computer. In this way, the compiler and not the programmer need be concerned with the hardware- dependent features of the computer.

* The program runs or executes quickly since it is now translated to machine language.

There are of course disadvantages to compiled languages:

Compiling a program is a multi-step process. The user-written program (called the source program) is entered and written as a file. This source file is then passed to the compiler.

If the compiler detects an error in the source program - perhaps a syntactical error - then the source program must be corrected and recompiled.

If after a program is compiled, an error is discovered during execution, the source program must be corrected and recompiled.

During the writing and testing of these programs, many hours might elapse going through these procedures. One alternative to the compiled programs was the development of interpreted languages.

6.3 INTERPRETIVE LANGUAGES

An interpretive language is one where the language elements are translated immediately into the machine language of the computer and executed. If a statement like **PRINT 3+5** is entered, the language interpreter would first check the syntax of the expression and then carry out the command step by step.

The command PRINT tells the interpreter that something must be displayed on the screen. If the something is a constant (literal), the interpreter knows that a quotation mark must follow the PRINT keyword. But if a mathematical expression follows the keyword, then the quotation mark will not be there.

The above example causes the interpreter to first calculate the sum of 3 and 5, and then display the result on the screen. The interpreter then proceeds to process any subsequent commands.

By using an interpreted language, the programmer can check his work at any time. This is in contrast to compiled languages where he has to first write the source file, compile it and then test it.

Because of the way that an interpreted language works, an interpreted program can never be executed as quickly as a compiled program can. But an interpreted language is considered a better alternative for the program developer.

His choice of language is based on a combination of the elements which a given language provides and the ease with which he can communicate with the computer.

6.4 HIGHER-LEVEL PROGRAMMING LANGUAGES

We have already seen that a number of programming languages have been developed in the course of the last twenty years. The purpose of a new language is either to open a new area of application for the computer, or to make the computer and its language more human.

There are languages, which are used to easily control machine tool operations. It would be impossible to write a bookkeeping package or even a video game in such a language, however.

The development of an all-purpose language was necessary for ease of learning, so BASIC, developed in 1964 at Dartmouth College, allowed beginners an easy introduction to programming. BASIC stands for Beginner's All-purpose Symbolic Instruction Code, which was an accurate description back when the set of commands consisted of only a few dozen instructions. But today, where commands are necessary for controlling graphics and sound processors, in addition to various methods of accessing files and even instructions for creating structured programs, the name "beginner's language" is no longer appropriate.

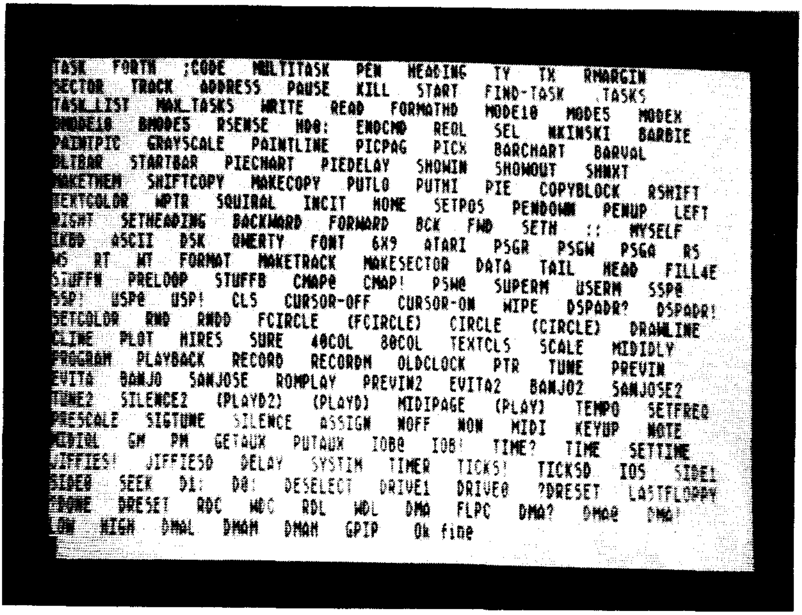
FORTRAN, BASIC and all other languages developed before 1971 are line-oriented programming languages. Each language statement is written on an individual line with an identifying line number. To use a routine of a program from more than one place in that program, you could use a GOTO instruction to the desired line number. As many programming textbooks maintain, the result was always something called "spaghetti code" - a very complex and chaotic execution path that makes editing and correction very difficult,' if not impossible.

Niklaus Wirth put an end to this by developing the programming language Pascal as a structured, easily readable and manageable language. He argued against "spontaneous" program development. In the Pascal language, the programmer must first define the variables and subroutines he will use. Pascal has been very successful as a teaching language at many American and European universities and schools, but has not met with the same success in industry.

The primary reason that Pascal has become so popular is that it has become available for almost every microcomputer. Likewise, the language Forth has also enjoyed wide success on microcomputers.

Forth is already available for the Atari ST series. See the following picture that displays this Forth's vocabulary.

The standard vocabulary is supplemented with words like PIECHART and BARCHART for easy implementation of business graphics. Other extensions such as SETFREquency, TEMPO, and BANJO make music and sound synthesis extremely simple.



```
TASK  FORTH  ;CODE  MULTITASK  PEN  HEADING  TY  TX  RMARGIN
SECTOR  TRACK  ADDRESS  PAUSE  KILL  START  FIND-TASK  .TASKS
TASK_LIST  MAX_TASKS  WRITE  READ  FORMATHD  MODE10  MODES  MODEX
BMODE10  BMODES  RSENSE  HD0:  ENDCMD  REQL  SEL  MKINSKI  BARBIE
PAINTPIC  GRAYSCALE  PAINTLINE  PICPAG  PICK  BARCHART  BARVAL
PLTBAR  STARTBAR  PIECHART  PIEDELAY  SHOWIN  SHOWOUT  SHNXT
MAKETHEM  SHIFTCOPY  MAKECOPY  PUTLO  PUTHI  PIE  COPYBLOCK  RSHIFT
TEXTCOLOR  WPTR  SQUIRAL  INCIT  HOME  SETPOS  PENDOWN  PENUP  LEFT
RIGHT  SETHEADING  BACKWARD  FORWARD  BCK  FWD  SETH  ::  MYSELF
?KBD  ASCII  DSK  QWERTY  FONT  6X9  ATARI  PSGR  PSGW  PSGA  RS
WS  RT  WT  FORMAT  MAKETRACK  MAKESECTOR  DATA  TAIL  HEAD  FILL4E
STUFFW  PRELOOP  STUFFB  CMAP@  CMAP!  PSW@  SUPERM  USERM  SSP@
SSP!  USP@  USP!  CLS  CURSOR-OFF  CURSOR-ON  WIPE  DSPADR?  DSPADR!
SETCOLOR  RND  RNDD  FCIRCLE  (FCIRCLE)  CIRCLE  (CIRCLE)  DRAWLINE
LINE  PLOT  HIRES  SURE  40COL  80COL  TEXTCLS  SCALE  MIDIDLY
PROGRAM  PLAYBACK  RECORD  RECORDM  OLDCLOCK  PTR  TUNE  PREVIN
EVITA  BANJO  SANJOSE  ROMPLAY  PREVIN2  EVITA2  BANJO2  SANJOSE2
TUNE2  SILENCE2  (PLAYD2)  (PLAYD)  MIDIPAGE  (PLAY)  TEMPO  SETFREQ
PRESCALE  SIGTUNE  SILENCE  ASSIGN  NOFF  NON  MIDI  KEYUP  NOTE
MIDIAL  GM  PM  GETAUX  PUTAUX  IOB@  IOB!  TIME?  TIME  SETTIME
JIFFIES!  JIFFIESD  DELAY  SYSTIM  TIMER  TICKS!  TICKSD  IOS  SIDE1
SIDE0  SEEK  D1:  D0:  DESELECT  DRIVE1  DRIVE0  ?DRESET  LASTFLOPPY
?DONE  DRESET  RDC  WDC  RDL  WDL  DMA  FLPC  DMA?  DMA@  DMA!
LOW  HIGH  DMAL  DMAM  DMAH  GPIP  Ok fine
```

These high-level languages can be divided into problem-oriented languages and procedure-oriented languages. Languages belonging to the first category give the programmer the means by which to solve special problems in terms that are familiar to the user. Members of the second category cover a broader spectrum of applications, where the programs are made up of a series of incremental procedures that solve the problem.

In the past few years, another group of languages has emerged which are growing in importance. These are the list-oriented programming languages; languages that work less with numbers and more with objects.

The most well-known of these languages is LISP for LISt Processor. LISP has been used especially in the area of artificial intelligence. This branch of computer science is concerned with describing human models of behavior to computers, to make them think and act like humans.

7. LOGO

Logo works with lists and is quite similar to LISP, which is why it is described as a subset of LISP by many researchers. As with Forth, it works with a stack and can be expanded with user-defined keywords. Like Pascal, procedures can be defined and variables can be declared as either global or local.

It's greatest advantage is the ease at which you can learn it. You can work with Logo after learning just three commands. You do not need to have a thorough overview of language to use it. Logo is therefore an ideal language for beginners although this does not mean that complex applications are not possible with Logo!

Digital Research has created a version of Logo for the IBM PC. Dr. Logo is one of the standard programs of the Atari ST computers!

Dr. Logo represents the most comprehensive implementation of this language available up to this time.

In the late 1970's, Logo caught the attention of the public with its turtle graphics. It was at this time that Dr. Seymour Pappert at the Massachusetts Institute of Technology developed turtle graphics to teach children about computers. Not only was he able to teach five and six-year olds about computers, but turtle graphics also enabled them to program the computer.

How else except with the help of a game could he win the interest of these children?

A child places himself in the middle of a room, and the other participants in classroom direct him using verbal commands so that he can trace mathematical figures with his movements.

A square might look like this:

go forward five steps
turn one quarter turn right
go forward five steps
turn one quarter turn right
go forward five steps
turn one quarter turn right
go forward five steps

By using a piece of chalk to mark the path he travels, a square is drawn. The result of the experiment can be discussed with the children and additional "games" of this type can follow. So the children can learn geometry by an entirely new method.

In the next step, the room is replaced by a computer screen and the subject is replaced by a graphic representation of a turtle. This makes it possible for children to duplicate actions in their real world on the computer--and since children like to draw, the computer becomes nothing more than an easy-to-use drawing instrument.

The picture of the square from the practice room can be immediately created on the monitor by every member of the group:

```
forward 50
right 90
forward 50
right 90
forward 50
right 90
forward 50
```

These children were able to see the result of their work. Their learning was reinforced by the immediate feedback provided by turtle graphics and Logo. Logo is extremely interactive. Instead of displaying cryptic error codes or a brief error message, it tells you exactly what you did wrong in a friendly way.

Logo is an ideal example of an interpreted language because Logo gives confirmation of the exectuion of the instruction for each input. You'll be able to quickly acquaint yourself with this language once you have your own Atari ST and Logo.

7.1 PROCEDURES IN LOGO

In Logo you can construct as many procedures as desired and thereby create a language tailored to your tastes, and application.

This process is not particularly complicated--you must think of an appropriate name for your new procedure and append *TO* to the procedure name. For example, the procedure **SQUARE** might look like this:

```
to square
forward 50
right 90
forward 50
right 90
forward 50
right 90
forward 50
end
```

To draw a square that is 50 units in length, you can tell the Logo interpreter to do this by typing the new procedure name: **SQUARE**.

But what if you want a square of a different size?

You could have the procedure draw a square of any size by using variables. Here's the amended procedure:

```
to square :length
forward :length
right 90
forward :length
right 90
forward :length
right 90
forward :length
end
```

In this case, a variable named *length* is introduced. The : tells Logo that it is a variable. Within the body of the procedure, the occurence of *:length* is replaced by the parameter that is tacked onto the procedure call. For example, *SQUARE 75* draws a square with a 75 unit side.

In the following pages, we'll take a brief look at the elements and predefined procedures in Digital Research's Dr. Logo.

1. Variables

Variables in Logo are designated by a prefixed quotation mark. Values are assigned to variables through the keyword *make*:

```
make "length 50
```

Logo recognizes both local and global variables. Local variables exist only within a procedure in which they are defined; once this procedure is exited, these variables are no longer accessible. The local variable **"length**, for example may be used to represent the number of characters of a word in one procedure and the dimensions of a square in a second procedure.

The instruction *local* makes an variable accessible to that procedure only: >local "length

Another property of Logo is its ability of to view variables as executable statements. This is very helpful since it's possible for the computer to program itself.

```
make "angle (forward 50 right 90 forward 40)
run :angle
```

The above example shows the method. A sequence of instructions is assigned to a variable which is then executed later with the *run* command.

2. Arithmetic operators

All programming languages have some method of performing arithmetic operations like addition, subtraction, multiplication, and division. Dr. Logo supports the usual operators +, -, * and /.

The usual notation for arithmetic operations is like this: 2+3

Dr. Logo also supports this notation:

Both yield a sum of five. The second notation is called *reverse polish notation.*

The functions sin and cos are available, where sine as well as cosine require as argument an angle given in degrees.

Other functions are:

int
returns the integer portion of a number by truncating it after the decimal point (the number is not rounded)

random
returns a random number between 0 and the value given as the parameter.

Example: *random 5* yields a number between 0 and 4.

3. Logical operators

Logical operators are not often used by beginners, but are by more advanced programmer.

The operators *AND*, *NOT* and *OR* are available. The logical comparators =, > and < are also available.

With regard to logical operations, Logo resembles Pascal rather than BASIC. In BASIC, the true condition is represented as -1, while Logo and Pascal return a TRUE.

4. Controlling the program execution

Programming in Logo is based on expanding the standard vocabulary. But Logo also requires commands for controlling the flow of a program's execution. In addition to the linear execution of program instructions, loops and branches are also possible. Here are the most often used:

bye
every Logo session should be ended with this command.

co
this command is an abbreviation for continue and causes a previously interrupted program to be continued at the point where it was interrupted

label ***name***
identifies a program line by *name*. This label may then designate that line as the destination of a jump by means of *go name* statement.

go ***name***
alters the flow of execution of the program; execution is continued at the line with the label identified as *name*.

if ***expression*** ***command-list-1***
command-list-2

evaluates *expression* and performs *command-list-1* if the expression is true; otherwise perform *command-list-2* on following line.

op

interrupts the current procedure without changing the input value. This is then returned as the result of the procedure.

repeat n

performs the command list enclosed in parenthesis n times.

E.g. repeat 4 (fd 50 rt 90)

run ***command-list***

executes *command-list* which follows.

stop

halts the execution of a program.

5. Graphics commands

With Logo, the user may divide the screen into different windows. The standard setting uses two windows, one of which is set up as a graphics window and the other for communication with the user. The size of the two windows can be changed, allowing the user to limit the field of movement of the turtle or to make the entire screen available for its movements. He can also eliminate the graphic screen and use the whole screen for text.

Commands for manipulating the text screen:

ct
erases (clear text) the text window.

pr (*list*)
prints the individual elements of *list* to the screen. The outermost parentheses are removed and the individual list elements are printed followed by a carriage return.

setsplit
sets the number of lines to be used for the text screen.

show (*list*)
similar to *pr*, but the outermost parentheses are not removed with *show*; *list* is displayed as the system sees it.

ts

set the screen to pure text mode.

type

similar to *pr*, but the concluding carriage return is not used.

Here are the commands for manipulating the graphic screen:

cs

clears the graphic screen and sets the turtle to its starting position.

clean

clears (erases) the graphic screen however the turtle remains at its current position.

dot (*x-coord y-coord*)

plots a point on the graphic screen coordinates x-coord, y-coord. The point is plotted in the current color.

fence, window, wrap

One problem in working with computer graphics when plotting a mathematical function is that a point may lie outside of the drawing surface. Most computers respond with an error message unless the program is designed to *clip* or remove those points lying outside the addressable coordinates.

While it is possible to move the turtle outside the drawing surface, it can then no longer be seen. The advantage of this is that instructions like forward 5000 do not cause the procedure to crash.

Wrap causes the turtle reappears on the left side of the screen if it disappears on from the right side. A movement of forward 400 in the direction north (beyond the top screen edge) brings the turtle back to its starting position (in the 640x400 display mode).

Fence restricts the turtle's movement to the actual coordinates. If the turtle tries to go beyond the edge of the graphics window, the message "Turtle out of bounds" is displayed.

fs

sets the entire (full screen) screen in graphics mode.

pal *colour-number*

sets the colour for the drawing pen to *colour- number.*

setpal

selects a palette of pens from the enormous number of colour shades which the Atari ST offers for drawing.

sf

returns information about the graphic screen. Includes window size, background colour, window status (window, wrap, or fence).

ss

with a split screen, the *ss* command switches back to a text window.

Here are the commands for controlling the turtle:

bk - back

fd - forward

these commands, followed by the length of the line in points, moves the turtle along this line in the current direction.

rt - right

lt - left

these two commands, followed by the angle of rotation in degrees, rotate the turtle at an angle in the appropriate direction.

ht - hide turtle

this command makes the turtle invisible during the drawing process. This speeds the drawing process since the computer no longer has to create and then erase the picture of turtle at each position.

st - show turtle

makes the turtle is visible once again.

Since all drawings are not made up of a single continuous line, the pen in Logo can be raised and lowered. The instructions necessary to do this are:

pu - pen up
pd - pen down

In addition, the pen can be replaced by an "eraser" in order to erase parts of the drawing. Do this with:

pe - pen erase
sets the drawing colour to the background colour so that the drawing can no longer be distinguished from the background. In addition, Dr. Logo offers a drawing pen which reverses the colour of the points.

px - pen reverse
automatically sets the unset points to the current pen colour.

The turtle may be moved to a new position directly:

setpos - set position
this command, followed by the x and y coordinates positions the turtle to a specific location on the graphic screen.

The direction of the turtle can be reset:

seth - set heading
this command, followed by a direction turns the turtle to an absolute direction in degrees.

The status of the turtle can be examined:

tf - turtle facts
this command returns information about the turtle. This includes the turtle's coordinates, heading, pen status and pen colour and indication as to whether the turtle is visible or not.

6. Primitives for word and list processing

ascii

returns the ASCII value of the first character of the following word.

char

returns the character corresponding to the following ASCII code.

bf - but first

returns all of the characters of the following word, except for the first character. For example, the function:

```
        bf "ATARI
```

returns: **TARI**

bl - but last

returns all of the characters of the following word, except for the last character. For example, the function:

```
        bl "ATARI
```

returns: **ATAR**

first

returns the first element of an expression. The element may be the first letter of a word or the first member of a list.

item *n*

returns the *n*th element of a list. For example, the function:

```
item 4 "ATARI
```

returns: **R**

count

returns the length of an expression. For example, the function:

```
count "ATARI
```

returns: **5**

If the expression is a list, the number of elements in the list would be returned:

```
count (monday tuesday wednesday thursday)
```

returns: **4**

emptyp *list*

returns either TRUE or FALSE depending on whether or not *list* is an empty list.

wordp *word*

returns either TRUE or FALSE depending on whether or not *word* is a word or a number.

word ***list***

returns the concatenation of two or more words. For example, the function:

```
word "super "computer
```

returns: **supercomputer**

list (*elements*)

returns a list composed of *elements* including the parentheses.

sentence (***elements***)

returns a list composed of *elements* without the outermost parentheses.

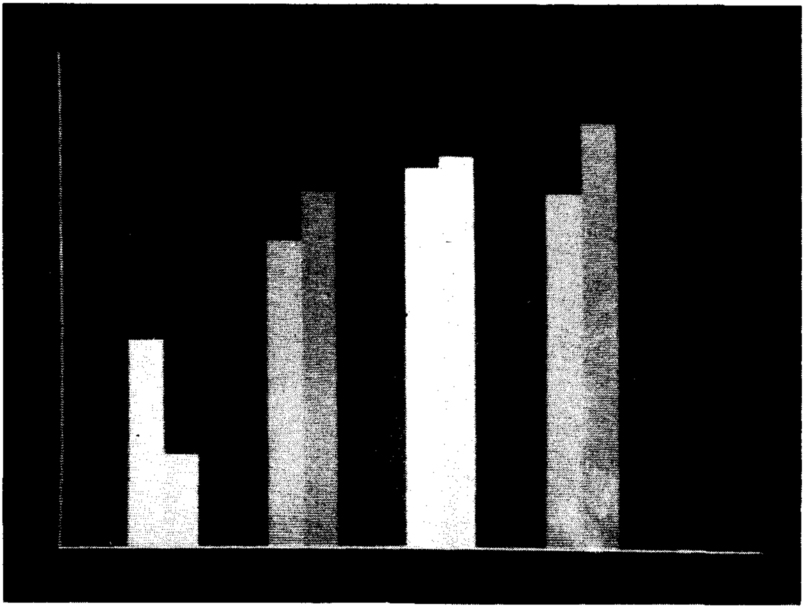
So you can see that Logo offers a large number of built-in procedures and functions. In addition to all of the pre- defined keywords, **Dr. Logo** also offers instructions for controlling the synthesizer chip and for reading the joystick as well as the mouse input.

An additional command:

.contents

causes Dr. Logo to display its entire vocabulary including primitives and any extensions the programmer has made.

Since the entire working memory can be saved to floppy or on hard disk, you do not have to redefine the procedures at the start of each session. By building upon earlier procedures,you can create new enhanced Logo systems, such as a special games Logo, or a graphics Logo which might include procedures like BARCHART and FUNCTIONPLOT, or even an intelligent system which can carry on discussions with the user.



7.2 SUMMARY

We hope that these pages have led you to the same conclusion that we found - that Logo is an excellent language for the beginner as well as for the advanced programmer.

A good programming language should be easy to learn. A few easy-to-remember expressions should suffice for simple tasks. You should not have to first master a large number of rules before defining variables.

The commands of the language should be easy to use. Its friendliness is enhanced if the it is simple to use the peripheral devices such as the printer or disk drive.

A truly practicable programming language should be easily expandable, so that extensions appear to be part of the standard commands. The user cannot distinguish between built-in and added commands.

A good language should be interactive so that the user can quickly develop and test new programs. In the event of program errors, it should display clear and helpful messages to the user.

All of these points speak for Logo. We should be happy that the Atari ST will offer both Logo and BASIC.

INDEX

Ex-Libris
Arthur Missira
MCMXCVII

First there was the fabulously successful VIC-20. Then came the record-breaking Commodore-64.

Now Jack Tramiel has launched his third home computer, the ATARI ST.

The ST promises to shatter all existing price- performance barriers and become a leader in the home-computer market.

This book, PRESENTING THE ATARI ST gives you an in-depth look at this sensational new computer that promises to bring you ..."**Power without the Price**"

1st
FIRST PUBLISHING LTD