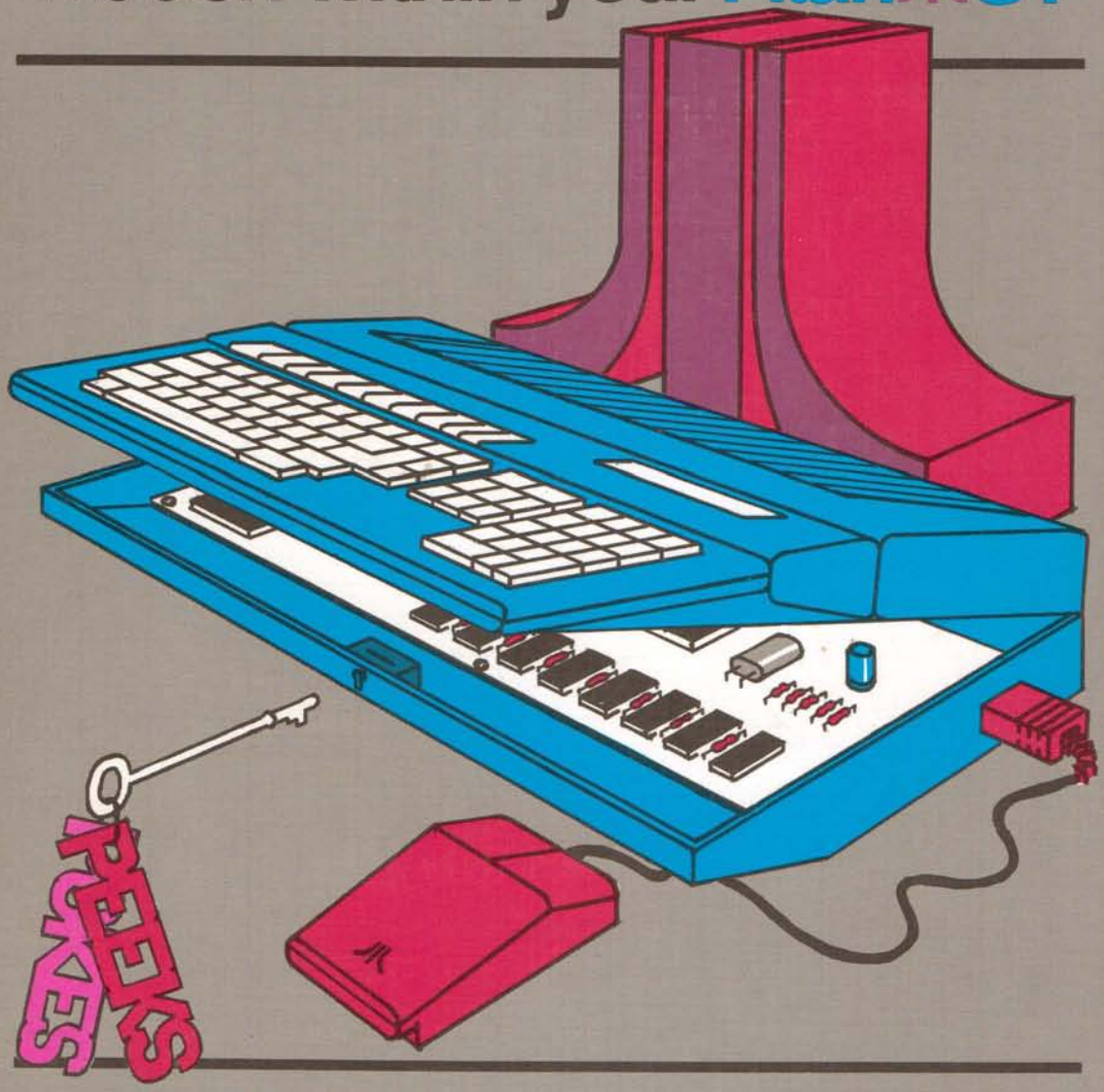
PEEKS & POKES

Keys to revealing the secrets hidden within your Atari® ST



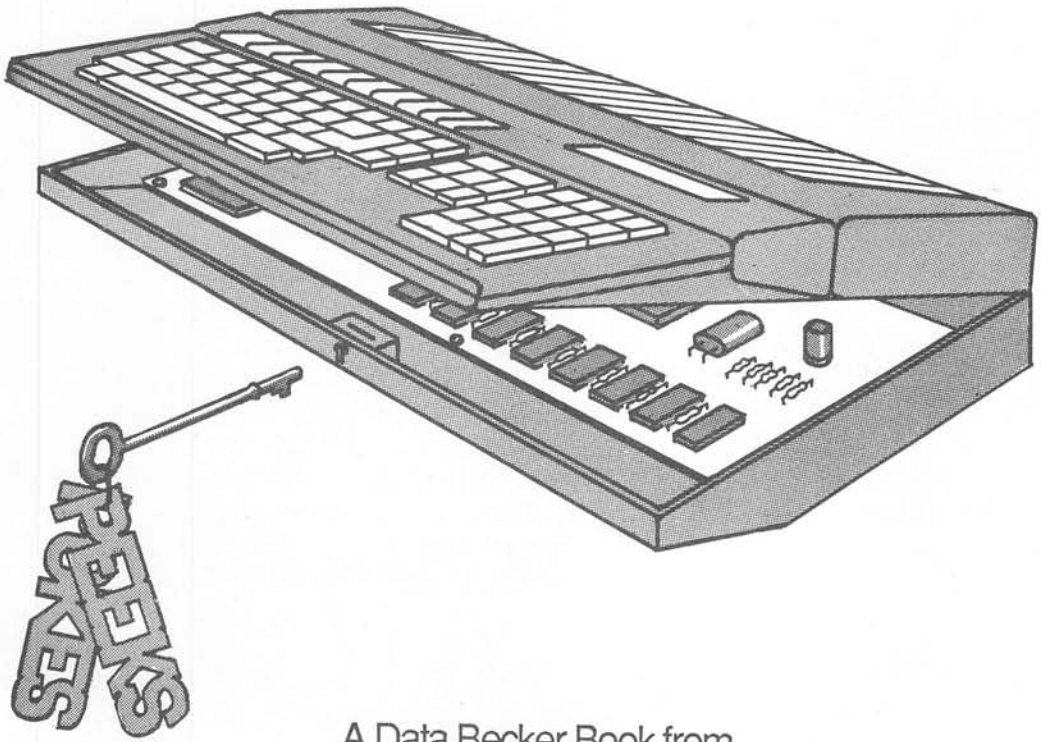
you can count on Abacus

A Data Becker Book

ATARI® ST™

PEEKS & POKES

Stefan Dittrich



A Data Becker Book from

Abacus

First Edition, October 1986
Printed in U.S.A.

Data Becker GmbH
Merowingerstr.30
4000 Dusseldorf, West Germany

Abacus Software, Inc.
P.O. Box 7219
Grand Rapids, MI 49510

ISBN 0-916439-56-9

Preface

The Atari ST offers new dimensions for the personal computer owner. It has great flexibility and ease of use at a very attractive price. Thanks to the user-friendly GEM operating system and mouse, the ST can be easily learned and used by the novice.

The purpose of this book is to give the ST user a closer look at many of the functions of his computer. We'd like to think of this as a "travel guide" through the fascinating world of the ST. On this trip you'll see some remarkable sights and many helpful signposts to help you find the right direction. The program examples will reveal many of the capabilities of the computer. And you'll be able to implement your own applications.

As you may gather from the title, we'll use the BASIC commands `PEEK` and `POKE` to investigate and influence the operation of the ST. Using these two commands and our knowledge of several internal tables containing important system parameters, we'll be able to manipulate the ST's operating system.

In addition, we'll take a look at the ST's communications capabilities. We'll discuss exchanging data with other computers, using a modem and connecting peripherals such as disk drives and printers.

Before we begin, there's one short note: Early ST computers were delivered with the operating system (TOS) contained on a diskette which was first loaded into the computer. Later ST's were delivered with the operating system contained on read only memory (ROM) chips. Depending on whether your computer has TOS on diskette or ROM, the addresses in some of these programs will differ.

These programs are written assuming that you're using an ST with TOS in ROM. The table in Appendix B can be used to find the corresponding addresses for the different versions of TOS.

Enough of an introduction. Let's start working with the ST. We present to you a collection of "quick hitters" for our favorite computer.

Thanks to my friends and associates for their cooperation, which gave this book an enhanced scope and added useful tips for you.

Stefan Dittrich

Hilden, West Germany
November 1985

CONTENTS

Chapter 1

A Look Inside the ST

A Look Inside The ST

A new computer is always inviting. It tempts you to sit down at the keyboard and start programming away. But by doing this, you may overlook the special features of the computer and treat it just like any other.

Treating the ST like any other computer would be a big mistake. Few other personal computers offer so many special features that will help even experienced programmers. This is not because of the powerful GEM user interface, which makes the ST user-friendly and easy to operate. It's also because the ST's hardware is so advanced that it can be used for nearly every personal programming application. For an overview, we have to look under the cover of the ST to view its individual components. Let's take a glimpse at the internal operations of this remarkable computer.

1.1. Internal Configuration

The outside of the ST doesn't indicate what is hidden inside. You might think, "That's really a nice-looking computer, but a little computer like that can't do much!"

The small housing of the ST (compared, for example, with the IBM PC®) hides an incredible amount of the latest technology. This technology gives the ST many new capabilities. Let's take a look at some of these new capabilities.

First there is the *central processing unit*, or CPU, which is the "brain" of the computer. This processor is the MC 68000 from Motorola. It is the largest component on the main circuit board. This processor has incredible specifications:

- 16 32-bit data and address registers
- 16 megabyte addressing capability
- 56 commands
- 14 addressing modes

The computer works internally with 32-bit words—in other words, with values up to $2^{32} = 4,294,967,296$. It has 16 data lines (this is the reason it's described as a 16 bit processor). These large numbers mean the computer can process a lot of data in a short time. This processor operates at 8 megahertz. This means it can execute up to eight million instructions per second. However, this is a theoretical value, since many instructions require several cycles. Nevertheless, the ST's speed is tremendous, as you can see during graphic presentations on the ST .

The "workers" in the ST are the additional chips that support the processor. There is an additional intelligent chip in the ST keyboard. It is called the HD 6301V1 and has an important job: it monitors all keyboard, mouse and joystick functions. Furthermore, this contains a clock which counts in seconds. For more information on the 6301 and the keyboard see Section 1.3, "The Intelligent Keyboard."

There are a few more special chips which make the ST the remarkable computer it is. The integrated circuits used were developed specifically for the Atari 520ST. The GLUE coordinates all the interactions of these ICs. Another chip is the MMU, or Memory Management Unit. This unit administers the working storage of 512K. The MMU is capable of handling up to 4 megabytes. The newest version of the ST, the 1040 ST, has 1 megabyte of working memory.

Next is the shifter chip. It controls the picture on the monitor. It also interprets the video memory in both monochrome and color modes, and sends the corresponding video signal to the monitor. The monitor connection of the Atari is located on the shifter. Video signals are available at the monochrome port, or RGB-signals at the red, green and blue ports. Information on the type of monitor attached is sent to the computer through the monochrome-detect port, which is set at either +5 volts for monochrome or 0 volts for RGB.

Another important chip is the DMA controller (DMA stands for Direct Memory Access). It accesses the ST's working memory directly and is used to transfer data to and from the disk drives or hard disk. The high speed of these data transmissions would overwhelm the CPU, so the DMA controller performs data transfer instead.

Control of the disk drive is the responsibility of the floppy-disk controller. This device starts the drive motor, moves the read/write head, obtains data from the diskette, and prepares the data for the ST. The

controller is capable of controlling either single- or double-sided disk drives. Also, the controller can handle either 3.5" or 5.25" disk formats. Programming the WD 1772 is very simple. It is supported very well by the operating system, so you won't have to worry about it.

Another very important IC is the MultiFunction Peripheral chip, the MFP 68901. Its specialty is input/output. It controls the printer port and the serial interface, and is internally responsible for timer functions.

Next we should mention the ACIAs (Asynchronous Communication Interface Adapters), which process serial data communications. One of the ACIAs handles the keyboard communications, while the other is responsible for the MIDI interface which operates at 31,250 baud. The MIDI interface can be used as a network connection.

The sound chip of the ST is a YM 2149. This IC is produced by the well known organ/synthesizer manufacturer Yamaha, and has three independent tone generators and a frequency generator. The envelopes, volume, filters and pitch are all programmable. The frequency range reaches from the lowest bass to almost ultrasonic frequencies well above human hearing levels.

All of these chips are programmable by the user. You can program the sound chip with the BASIC commands `SOUND` and `WAVE`. Each IC has an area of memory reserved for its exclusive use in which it stores its variables and parameters. Information exchange takes place in this reserved memory. We'll go into more detail on this later in Section 2.1.

1.2 Interfaces

The value of a computer in part, is determined by its ability to communicate with the "outside world." This communication is handled through its *interfaces*.

Data transfer from a sender to a receiver is accomplished in one of two ways:

- Parallel
  or
- Serial

1.2.1 The Parallel Interface

Parallel data transfer means that the entire byte is sent to the receiver at once. The advantage of this method is the high speed of transmission. The disadvantage is that it requires eight "wires", one for each data bit.

A typical example of a parallel interface is the Centronics port. Usually it is used to connect to a printer, since it can only transmit data in one direction, i.e. from the computer to the printer. The Centronics port is located on the back of the ST and is labeled by a printer pictogram.

The physical connection between this plug and the printer (Epson, for example) is through a cable with at least 11 wires, as follows:

Computer Pin	Wire	Printer Pin
1	Strobe -------->	1
2	Data 0 -------->	2
3	Data 1 -------->	3
4-9	Data 2-7 ------>	4-9
11	Busy <-------	11
18	Ground	19

As you can see, the busy signal line is connected to the computer. This return line is called the *handshake line* and is required to ensure the reliability of the data transmission.

If the computer wants to send a character to the printer, it places the ASCII value of the character (0-255) in binary at the data port and informs the printer through the LO signal at the strobe that it can accept this data. The printer accepts this data into its memory. If it is capable of accepting additional data, the printer informs the computer by keeping the Busy Line LO. If the printer cannot accept additional data, it sets the Busy Line HI. When the printer is ready to accept more data it turns the Busy Line back to LO.

If the printer is not connected or is not online, the computer receives no response on the busy line. It waits for a while until it finally gives up and, in certain cases, produces an error message. This timeout on the Atari ST is rather long—it takes 30 seconds. The reason is that the ST doesn't use the Centronics ACK (Acknowledge) signal line, which would permit the computer to immediately recognize if the printer is ready. Since the printer could be tied up at that moment, sufficient time must be provided for the printer to complete its current activity.

The ST offers a special feature in this interface which is not commonly found on other computers. It can read from the parallel port. The application possibilities of this are enormous. This makes it possible to connect digital circuitry and to evaluate the signals provided. Interrogation of these signals from the printer port occurs through the BASIC command `INP(0)` which outputs the available data byte. Through this procedure the ST can accept data which is really intended for a printer. For this operation only the Centronics port of a computer must be connected with the parallel plug of an Atari where the connections 1 and 11 are crossed at the connection. After doing this the following small BASIC program can be run on the ST:

```
10   rem*** Printer - Simulation 1.2.1***
20   x = inp(0)          : rem Data input
30   print chr$(x);      : rem output char.
40   goto 20             : rem Infinite Loop
```

All printer output from the other computer will now be output to the output window of the ST. We can transfer data between computers without RS-232 ports using this method.

In addition, the ST can also be used to control machines. A good example are the robotic kits offered by Fischer-Technik for computers. Small robot arms or simple plotters can be built from these sets, which can then be controlled by the ST. To run such an application, the computer must be able to accept and evaluate data from this peripheral and then provide control for the device. These sets have a serious purpose, since computer-controlled machines in industry operate on the same principles. The ST is theoretically capable of controlling a machine tool.

1.2.2. The Serial Interface

The serial interface transfers data, bit by bit. Therefore only one line is required—but, ironically, serial data transmission is more complicated than parallel transmission.

The serial interface of the ST is the 25 pin connector, labelled with a telephone symbol, found on the back of the computer. This is a standard RS-232, interface that can either send or retrieve data. Pin 2 of the connector outputs data (TD = Transmitted Data). The HI/LO signals are sent as + or -12 volts.

Here is the pin description for the RS-232 interface:

1 CHASSIS GROUND (shield)
This is seldom used

2 TD
Transmit data

3 RD
Receive data

4 RTS
Ready to send comes from I/O port A bit 3 of the sound chip and is always high when the computer is ready to receive a byte. On the Atari, this signal is first placed low after receiving a byte and is kept low until the byte has been processed.

5 CTS
Clear to send of a connected device is read at interrupt input I2 of the MFP. At the present time this signal is handled improperly by the operating system. Therefore it is possible to connect only devices which "rattle" the line after every received byte (like the 520ST with RTS). The signal goes to input I2 of the MFP, but unfortunately is tested only for the signal edge. You will not have any luck connecting a printer because they usually hold the CTS signal high as long as the buffer is not full. There is no signal high as long as the buffer is not full. There is no signal edge after each byte, which means that only the first byte of a text is transmitted, and then nothing.

7 GND
Signal ground.

8 DCD
Carrier signal detected. This line, which goes to interrupt input I1 of the MFP, is normally serviced by a modem, which tells the computer that connection has been made with the other party.

20 DTR
Device ready. This line signals to a device that the computer is turned on and the interface will be serviced as required. It comes from I/O port A bit 4 of the sound chip.

22 RI
Ring indicator is a rather important interrupt on I6 of the MFP and is used by a modem to tell the computer that another party wishes connection, that is, someone called.

The designation Modem indicates that this plug is intended for connection of a telephone modem. A modem provides the connection between the computer and the phone line by converting the HI or LO signal into two separate tones. If you want to transmit data to a person who has a computer with a telephone modem, you simply call him. When both of you have activated your modems, the data transmission can begin.

To signal the receiver that a data word is starting, a *startbit* is transmitted at the of each word. The signal goes to HI at first; next the data is transmitted, concluded with a LO signal as a *stopbit*. The speed at which

the data is transmitted is called *baud rate*. This number (for acoustic modems 300) indicates the number of data bits transmitted per second.

If you don't know anyone who owns a modem, you can contact a bulletin board service (BBS). These online information exchanges are offered by many private and public agencies, and even individuals. The phone numbers are available in trade papers, magazines, and by word-of-mouth.

1.2.3 Disk Drive Connections

The large round port on the back of the ST identified by a floppy disk drive picture represents the connection for one or two disk drives. It is controlled by the WD 1770 disk controller, which converts the operating system commands into electronic signals that the disk drive can respond to. If, for example, you want to see the directory of a diskette, GEMDOS asks the controller to provide the data from the corresponding sector of the selected diskette. The controller has to perform these steps :

1. Select the desired unit (1 or 2)
2. Start the disk drive motor
3. Position the read/write head on the track containing the sector
4. Wait for the 'index pulse' signal that is generated once during each revolution and tells the controller the current diskette position.
5. After reaching the appropriate sector, reads and converts the analog signals from the read/write head into digital signals.
6. Transmit the signals to the ST memory.

For double-sided drives it must also specify which side of the disk to store the data on. This signal comes from the sound chip(!). The standard floppy unit, SF 354 drive, which writes 500K in unformatted condition,

writes only on one side. Double-sided units are fully supported and can be connected directly, however. The pin assignment is as follows:

Pin	Name	Significance
1 (30)	Read Data	Carries the signal from the read/write head
2 (32)	Side 0 Select	Selects which side of diskette to use
3 (3)	GND	Ground (0V)
4 (8)	Index Pulse	One signal per revolution
5 (10)	Drive 0 Select	Selects drive 0
6 (12)	Drive 1 Select	Selects drive 1
7 (33)	GND	Ground
8 (16)	Motor on	Turns motor on
9 (18)	Direction In	Determines direction of read/write head
10 (20)	Step	Moves one step in or out
11 (22)	Write Data	During write data signals from the computer
12 (24)	Write Gate	Signal permits writing data
13 (26)	Track 0	Announces that outermost track has been reached
14 (28)	Write Protect	Signals that the diskette is write protected

The signals are standardized and correspond to the bits of the Shugart connections. It is therefore easy to connect other disk drives.

Caution: a wrong connection can destroy the controller! The equivalent pin numbers of the Shugart connector are in parentheses in the above table.

Before a diskette is used for the first time it must be formatted. Choosing the desktop menu 'Format' and the prompt OK will give you a choice of formatting your disk as either single- or double-sided. With a single-side unit you should never select 'Double-Sided', but you can format both single- and double-sided disks with a double-sided disk drive. During formatting, the tracks on the diskette are erased and individual sectors are determined. Be careful: all data which had been on the diskette is erased!

1.2.4 The MIDI-Interface

On the back of the ST are two 5 pin DIN (Deutsch Industry Norm) ports. They are identified as MIDI-In and MIDI-Out and support electronic musical instruments such as synthesizers.

How does this work?

MIDI stands for Musical Instrument Digital Interface and this interface has been built into many music synthesizers. If the ST is connected with one or more synthesizers (up to 16), the ST becomes a musician that is capable of recording melodies being played or controlling the instruments. The information exchange is serial, following the same principle as the modem port, but the transmission speed is a rapid 31,250 bits per second.

Since several instruments can be attached to the MIDI interface, it's necessary to be able to differentiate between the units. This is done with the select line, just as in the selection of disk drives. The receivers are all attached in parallel to the same wire. The computer first sends a signal which is recognized by only one unit, then sends out the desired command to that device.

This fast interface is predestined for use in applications of a different kind. Other computers can be connected to the ST through the MIDI lines to construct a network of up to 17 computers. Each one of these computers must have its own MIDI interface and an identification number. Communications can be performed in BASIC if high speed is not important. Output can be performed with the `OUT 3,X` command and input with the command `X=INP(3)`. The 'X' is the value of the data to be transmitted.

1.2.5 The ROM Expansion port

On the left side of the computer is another less obvious connector port. It carries no designation, but accommodates a 40 pin cartridge. Such a cartridge can be equipped with ROM memory up to a maximum of 128K. Possible applications are enhancing the operating system, utility

programs that can catch a system crash before it happens, or simple games. The large number of connections required in this port is due to the fact that it must address 128K of memory using a total of 17 bits, and also that transferring the data requires a total of 16 additional bits.

This plug is apparently missing a read/write line which permits data output. You cannot write data into the address area of the connector! If you do not have something important stored in memory at this time you can try it. The connection lies between the addresses $FA000 to $FC000 (1024000 - 1032192). If, for example, with TOS loaded from diskette you enter `POKE 1030000,0` the system will crash and you will find yourself in the desktop. You will have to press the reset button to restart.

The expansion plug is intended only for use with ROM expansion cartridges, similar to those used with the old Atari 400/600/800 computers (Those cartridges do not work with the ST). The first data on the cartridges contains identification numbers which tell the ST what type of programs are provided on the cartridge. If the first long word at address $FA0000 is equal to $FA52255F, it is a diagnostic program. If it is $ABCDEF42, an application program is contained on the cartridge. All other combinations are ignored. If a diagnostic program is contained, the processor starts with a system reset almost instantly at address $FA0004. Application programs must store various information at this address including starting address, length and name of the program.

1.2.6 The Mouse/Joystick Connection

The two plugs on the right side of the ST are for joysticks. Plug 0 can also be connected to the mouse. The main difference between joystick and mouse lies in the joystick having a signal pin for every one of the four directions on which it can place a LO signal (0V). The mouse, on the other hand, provides further direction impulses to the system, since it can be moved at various speeds.

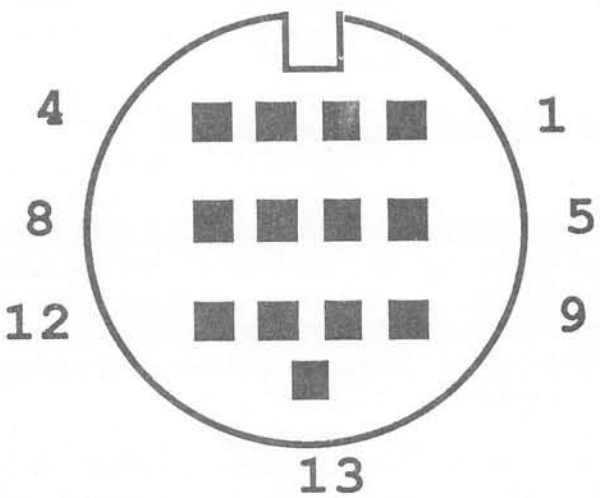
Joystick and mouse connections are quite similar. Joysticks can be attached to either port, but the mouse can be operated only on plug 0. Since the mouse has two buttons, this connection has one additional input.

The mouse/joystick plugs are only for input. The keyboard processor does not have a command to change the direction of data transfer. It is possible but impractical, to replace the processor with a unit whose operating system would recognize additional commands. Output signals are easily routed through the parallel port (printer), while this plug is also well suited for input signals. Digital technology often uses the signal voltages 0V (LO) and +5V (HI); these correspond to the TTL level (TTL = Transistor-Transistor-Logic). To read a key from the computer, it only has to be connected to ground (Pin 8) and one of the pins 1-4.

1.2.7 The Monitor Connector

A monitor lets you see what the computer is doing. The Atari ST provides a connector for a monitor. The ST has a connector for using either the SM124 monochrome monitor or the SC1224 color monitor.

The newer 520 ST also has a connector for attaching a television set. Below is the pin layout of the monitor connector:



1.3 The Intelligent Keyboard

The micro-processor built into the keyboard has many tasks to perform. It takes a load off the main 68000 processor by a constant surveillance of the mouse and keyboard. With its own memory, the keyboard processor is capable of monitoring the status of the mouse, keyboard and joystick and making this information available to the 68000 at any time on demand. The main processor does this by sending a serial command and parameters to the keyboard processor, which returns the desired information. This happens at a speed of 7,812.5 bits per second. The keyboard processor is programmed so that it can perform the commands following:

1.3.1 Command Overview

Command	Reaction
7	Set flag when mouse key is pressed
8	Immediate report on relative mouse position
9	Immediate report on absolute mouse position
10	Report mouse movement as cursor key activation
11	Delay report of mouse movement
12	Set scale for mouse position
13	Read absolute mouse position
14	Set internal coordinate system
15	Y coordinate jump is below
16	Y source is above
17	Resume transmission of data (after #19)
18	Turn off mouse
19	Stop data transmission
20	Immediately report all joystick movement
21	Turn off function 20 as well as the mouse
22	Transmit joystick position
23	Sense joystick continuously
24	Continuously sense joystick key position

25	Continuously sense joystick movement as cursor key activation (joystick 0)
26	Turn off joystick
27	Set time of day
28	Report clock time
32	Load keyboard storage
33	Read Keyboard storage
34	Start a program in keyboard processor

Furthermore, it understands various inquiries about the current conditions of the status.

Let's look at a few examples of the applications of the keyboard commands. The problem in programming lies with the different number and significance of the parameters which must be passed for individual functions. Furthermore, it is not always easy to obtain the results of the functions. Let's consider an example for this. Function 22 causes the keyboard processor to report the current condition of joystick 1. The result is stored in memory location 3592 with TOS in ROM (see Appendix B for TOS in RAM.) A program for this would appear as follows:

1.3.2 Reading the Joystick

```
10   rem ***Joystick 1 - Sensing 1.3.2 ***
20   out 4,22              : rem Function call
25   defseg = 1      : rem peek one byte
30   js = peek(3592)and 255 : rem result
40   print js              : rem Print value
50   goto 20         : rem Reread joystick
```

Using the BASIC command `OUT 4,n` the keyboard senses the condition of joystick 1 and reports it to the central processor. The operating system stores this information in location 3592, where it can be read with the `PEEK(3592)` function. This value represents the joystick position in binary. Bits 0 and 1 indicate the vertical motion of the joystick. Bit 2 and 3 are for the horizontal direction. As you can see, the function does not require a parameter. This is illustrated with the next sample program.

1.3.3 Mouse as Cursor Control

```
10    rem **Mouse Movement as Cursor Keys 1.3.3 **
20    out 4,10              : rem Send command
30    out 4,10              : rem X-Delay
40    out 4,15              : rem Y-Delay
50    rem out 4,8: rem turn mouse pointer on
```

To see the results of this program: Enter this program, run it and then enter the Editor and move the mouse.

Here we're telling the keyboard to output the movement of the mouse as if the cursor keys were being used. This is useful in BASIC program editing, since this program allows you to rapidly move the cursor across the screen. The delay values in lines 30 and 40 signify the horizontal or vertical movement of the mouse for each cursor step. You should not use large values if you have only a moderately large desk. To see the mouse again as a pointer, enter the command `OUT 4,8`. As illustrated, the values given are entered as `OUT` commands. You must be very careful to maintain the correct parameter number, because an incorrect parameter can stop the data transmission between keyboard and the computer.

1.3.4 Time and Date Functions

Now we come to an example in which parameters are both input and output. We are talking about the clock, which is controlled in the keyboard. This has many interesting applications.

```
10    rem *** Read Time 1.3.4 ***
20    out 4,28          : rem Check Time
30    for i = 3584 to 3589: rem Time/Date Buffer
40    x = peek(i) and 255 : rem Read double number
50    a$=a$ + str$(int(x/16)): rem First number
60    a$=a$ + right$(str$(x and 15),1)
70    a$=a$ + ":"
80    next i
90    print "Date    :"; Left$(a$,12)
100   print "Time    :"; right$(a$,12)
```

```
110  end
130  rem *** Set Time/Date ***
140  out 4,27          rem Set Time command
150  print "Please input Date/Time
     (YYMMDDHHMMSS)"
160  input a$          : rem Input
170  for i = 1 to 6 : rem BCD Calculation
180  d =0
190  d =d + asc(mid$(a$,i*2,1))-48
200  d =d + (asc(mid$(a$,i*2-1,1))-48)*16
210  out  4,d          : rem Set Parameter
220  next i
230  a$="" :goto 10 :rem Output new Time
```

Actually these are two separate programs. Program 1 displays the actual date and time. The format is year, month, day, hour, minute, second. The use of the format in the text variable a$ is completely up to you. Furthermore, the function can be made into a subroutine by changing the END statement in line 110 to RETURN. The most complicated part of the program is the change of the BCD form (Binary Coded Decimal) into individual numbers. In this coding method, each date number only uses 4 bits of a data byte. For example, the value of the BCD number 31 is $31, which corresponds to a decimal 49 (See also chapter 2.1).

The same number occurs in the conversion of single numbers. In Line 130 the routine begins to set the date. After calling the 'Set Time' command (function 27), the time is entered. For example, 053010192015 stands for the date October 30,1985 at 19:20 and 15 seconds. The 8 is automatically added for the year. This value is automatically converted to BCD numbers and passed to the keyboard. The statement goto 10 in line 230 serves to determine if the function was correctly executed, and can be omitted.

1.3.5 Reading Keys

We should also discuss an important part of the keyboard, namely the keys themselves. The keyboard is constantly read by the built-in processor. If a key is pressed, the value of this key is stored in its own memory. The memory is limited to 128 bytes. It also contains much other information, but this is sufficient. The main processor, i.e. the operating system, obtains the actual data in a cyclical manner. The keys activated are then stored in the main memory and read out when required.

If you want to read a key with a BASIC program, there are several methods to use. You can wait for a keypress with the statement `X=INPUT$(1)` to store the result in `X`. This function provides the ASCII code of the normal alphanumeric keys. The information obtained with the statement `X=INP(2)` is much more comprehensive. The `X` value obtained in this manner contains the actual key code from the key that was pressed. Included are the function keys which have the values 187-196. If you want to wait for a function key to be pressed, you can use the following program:

```
90    rem*** Fkey .bas 1.3.5 ***
100   print "Please press a Function Key!"
110   x = inp(2)       : rem Sense Key
120   if x < 187 or x > 196 then 110:rem Wrong Key
130   ft = x - 186     : rem F-Number
140   rem on ft goto f1,f2,f3,f4,f5,f6,f7,f8,f9,
      f10
150   print "You pressed function key F";ft
```

If you remove the `REM` from line 140, this program will branch to the selected program section after you press a function key. You can give these sections almost any name you want instead of F1 - F10. This type of programming is used for menu control.

1.4 The Mouse

The mouse with its two buttons is a fast and convenient input device. Every movement of the mouse is registered onscreen in X and Y steps with a resolution of four steps per millimeter. How does this work?

A look into the inner housing of the mouse gives some answers. If you remove the ball by sliding the cover away on the bottom of the mouse, you see three small metal rollers. The large roller serves to stabilize the ball. The others transmit the corresponding rolling motions through a perforated disk which has two light sensors.

The processor chip in the keyboard, the HDV 6301 V1, evaluates the four impulse sequences. Assuming you move the mouse forward in the Y direction, the processor obtains two opposing impulse sequences at the Y input. At first, the processor waits for both direction signals to be set HI, and recognizes the direction the mouse has been moved by the drop of the first of the two signals to LO level. The same procedure is applied to to movement in the X direction, so that the actual direction of the mouse movement can be recognized.

The processor in the keyboard transmits the direction of the mouse movement, keypresses, joystick position, time and date at the request of the operating system. The transmission occurs at 7,800 bits per second (baud). Fortunately, you don't have to concern yourself with the data transmission, since the mouse position is detected by several programming languages. The DR LOGO delivered with the computer offers the MOUSE command which returns a typical LOGO list.

1.4.1 The Mouse as a Paintbrush

A sample LOGO program for drawing on the display (graphic window):

```
TO DRAW
 IF (ITEM 4 MOUSE) [SETPOS MOUSE]
 DRAW
 END
```

This program is entered with the LOGO editor and called later in the command level with DRAW.

This is a *recursive program*—that is, the program can call itself. This is accomplished by the command DRAW at the end of the procedure. The second line shows the syntax required for the LOGO command MOUSE. This can be regarded simply as a variable with values listed one behind the other. The sequence of the individual entries in this list is [x y B1 B2 B3], where x and y are the current coordinates of the mouse pointer. B1 and B2 signify the status of the two keys on the mouse–B1 for left and B2 for right. They are TRUE when the corresponding key is pressed. B3 indicates if the mouse indicator is located in the graphic window of LOGO (TRUE when yes). In the above sample program the fourth entry (ITEM 4 MOUSE) shows the use of the right cursor key. With this instruction a line is drawn to the mouse indicator when the right mouse key is pressed.

When you run the program, you will find that nothing happens if you move the mouse while pressing a key. This also occurs in BASIC programs. The reason for this is that GEM must draw the mouse pointer continuously while the mouse is moving. This keeps the computer from doing anything else. This side effect can be useful if you want to stop a program temporarily, and roll the mouse around while you read the output values.

Not every program is as simple as those in LOGO. For example, BASIC does not offer a command for reading the mouse. You have to perform some programming gymnastics with the VDI in GEM (see section 4.2: GEM).

One tip for owners of an older model Atari or a similar home computer: the plug on the mouse can be inserted into any joystick port. Through a very fast machine language program, the mouse can be connected to your old machine and used for input. The reading of the mouse inputs must occur about 1,000 times per second, since the conditions change with a movement of 1/4 millimeter!

Chapter 2

Memory Structures

Memory Structures

All data input such as programs must be stored in some manner, whether in RAM/ROM or on magnetic mass storage devices such floppy disks. The different capabilities of these storage devices are discussed in this chapter.

2.1 Internal Memory

There are two different types of internal storage:

- The read/write memory (RAM = random access memory)
- The permanent memory (ROM = read only memory)

The Atari 520ST has a RAM of 512K, which is exactly 524,288 bytes. The Atari 1040ST contains double that amount, namely 1024K. These electronic circuits retain their contents only when supplied with current. If power fails during a work period, the entire contents of the internal memory are lost. To guard against this unpleasant accident, you must store items by occasionally writing them to a disk.

RAM and ROM contain all data and programs that the computer needs to function. In the current version of the Atari ST, the operating system of the computer is in ROM, so it doesn't take up any RAM. However, the screen memory uses 32K. In addition, memory is used for tables and constants for the operating system. Additional RAM is used if a programming language such as LOGO, which is about 110K long, is also loaded.

This read/write memory can be changed at will. The data mentioned above, which the operating system requires to store system parameters, can be manipulated at any time. This is a great challenge for all programmers who wish to step beyond the bounds of normal programming to fully utilize the capabilities of the system. The BASIC commands `PEEK` and `POKE` let you accomplish this. Before trying anything though, you should know what is available in the various memory locations.

Caution: Manipulation of unknown memory addresses can have unexpected effects which become noticeable only later!

Let's examine the contents of the Atari ST in more detail now.

From the variable memory locations, we now come to the fixed memory locations—the ROM (Read Only Memory). These are permanently programmed ROM chips whose contents remains intact, even without power. In the new version of the ST, the operating system TOS is installed in ROM so that it is available immediately after power-on. In the earlier computers of the first series, the ROM contained only a program which loaded TOS from the disk. The only advantage of the disk based TOS is the capability of altering the operating system.

Let's examine the entire memory area of the ST. Since the computer has an address bus of 24 bits, it can directly address 2^{24} or 16 megabytes. This enormous addressable area is not completely occupied with memory chips. The following table shows the memory assignment of the computer.

2.1.1 Address Assignment of the ST

Range	Type	Purpose
00 0000	ROM	Reset: supervisor stack pointer
00 0004	ROM	Reset: init vector
00 0008	RAM	RAM start
07 FFFF	RAM	RAM - top with 512K
0F FFFF	RAM	RAM - top with 1 megabyte (520ST+)
1F FFFF	RAM	RAM - top with 2 megabytes
3F FFFF	RAM	RAM - top with 4 megabytes (maximum)
40 0000 F9 FFFF	Unused	Unused area

FA 0000 FB FFFF	ROM	ROM addition (128K)
FC 0000	ROM	ROM operating system (192K)
FE FFFF	ROM	
FF 0000 FF 7FFF	Unused	Unoccupied area
FF 8000 FF 8800	I/O	I/O area (2K)
FF A000 FF BFFF	I/O	I/O area (2K)
FF C000 FF FFFF	Unused	Unoccupied area

As you can see, RAM lies in the area from 00 0000 to 07 FFFF; and the operating system permits extension up to 4 megabytes. The operating system itself is stored (later) in ROM, which starts at location FC 0000. Higher up are the areas for I/O (Input/Output) chips (see Section 1.1). The addresses in which parameter transfers take place with special IC's are in this mempry area. Access by the user is immediately stopped by a program interrupt. The BUS-ERROR leads to the output of the much-dreaded cherry bombs on the screen and to the return of the GEM desktop. This I/O area is arranged as follows.

2.1.2 The Addresses of the I/O Chips

from FF 8000	• memory configuration
from FF 8200	• register for the video chip
from FF 8600	• DMA/disk-controller
from FF 8800	• register for the sound-chips
from FF FA00	• timer and interrupt chip
from FF FC00	• keyboard and MIDI-chips

One more important area must be mentioned: the first kilobyte of memory beginning at address 000000. Here important system parameters are stored and normally the user is not permitted to manipulate this area

(or you see cherry bombs!). These parameters are called *vectors*, i.e., in these memory locations are the jump addresses the computer branches to when program interruptions occur. These interruptions are called *exceptions* and are caused for various reasons. For example, the computer gains access to protected areas of memory (I/O or also other vector/tables) by using these exception vectors. The table is organized as follows.

2.1.3 Error Vectors

Number	Address	Used By
	$000	Reset: initial SSP
	$004	Reset: initial PC
2	$008	Bus error
3	$00C	Address error
4	$010	Illegal command
5	$014	Division by zero
6	$018	CHK command
7	$01C	TRAPV command
8	$020	Privilege violation
9	$024	Trace
10	$028	Axxx - command emulation
11	$02C	Fxxx - command emulation
	$030-$038	Reserved
	$03C	Uninitialized interrupt
	$040-$05F	Reserved
	$060	Unjustified interrupt
	$064-$083	Level 1-7 interrupt
	$080-$0BF	TRAP command
	$0C0-$0FF	Reserved
	$100-$3FF	Interrupt table
	$100	Parallel port internal
	$104	RS-232 carrier detect
	$108	RS-232 clear to send
	$10C	Unused
	$110	Unused
	$114	200 Hz system clock
	$118	Keyboard/MIDI interrupt

$11C	Unused
$120	HSync
$124	RS-232 transmission error
$128	RS-232 transmit buffer empty
$12C	RS-232 receive-error
$130	RS-232 receive buffer full

Not every vector is used, but many are used quite often. The number of the vector corresponds to the number of cherry bombs that are displayed. After a system crash, you can guess the cause simply by counting them. When such a crash occurs, the operating system tries to recover as best as possible. Often it does not succeed, and you must push the reset button. To resume operations, the computer requires the information about the conditions before the crash. This information was saved by the operating system in the following locations:

$0380	=$12345678, if data is valid
$0384	from here on are stored the D0-D7 registers
$03A4	from here on are stored the A0-A6 address registers
$03C0	the old supervisor-stack-pointer A7
$03C4	Number of failures that occurred
$03C8	Here is the old user-stack-pointer A7
$03CC	and 16 words from old supervisor-stack pointer

The type of declaration of the vectors and data types may be unfamiliar to you, since it concerns procedures in the machine language area. For an explanation of the procedures in machine language and how to use them, refer to the Abacus book *Atari ST Machine Language*. But first, what are these vectors? We shall examine them closer.

2.1.4 Pointers

Using pointers, also called *vectors*, is a very important technique in writing computer software. A pointer is a memory location at which an address has been stored.

An example is the pointer located at $44E. The address of the video display memory (video RAM) is stored in this pointer. By referring to

this pointer, the BIOS and the GEM can determine the location of video RAM. Thus a program can address, through a pointer, the memory areas whose location may differ for various models of the ST.

Here's another example. To access the video RAM, we need only read the value of this pointer and use it as an address.

```
10   rem *Set display point with POKE 2.1.4*
20   defdbl a: p=1102: rem 1102 = $44E
25   a=peek(p)
30   z =20              : rem Display line
40   s =5               : rem Display column
50   bz =peek(10556)      : rem Byte per line
60   for i = 0 to 10      :rem Start of loop
```

Lines 20 and 25 show one of the "tricks" in using pointers from BASIC. You may recall that an integer variable normally yields 16 bits of precision. Since the memory addressing range of the ST far exceeds the 64K of a 16-bit value, we must use a 32-bit integer.

The operating system does not have to go through this process, since the processor can work directly with long words. The composition of the desired display address takes place in lines 50 and 60. A character is really a matrix consisting of 16 x 8 points. You can write a character on the screen as follows, but let us first change the above program slightly:

```
70   read x$            : rem Read sample line
80   x=0 : for j = 1 to 10: rem Evaluation
90   x= x-(mid$ (x$,j,1)="*") * 2^(10-j)
100  next j
110  poke a + s + (z+i)*bz,x: rem Read line
115  print x
120  next i
130  end
135  rem --- Sample Data ---
149  data "            "
150  data "    **      "
160  data "   ****     "
170  data "  ******    "
180  data " ** ** ** ** "
190  data "**********"
200  data "** **** **"
```

```
210   data "**   **   **"
220   data "  **   **   "
230   data "  ******   "
240   data "    **     "
```

Now we have bypassed the operating system and drawn a smile on the screen. But what happens if the value of the vector is changed? Let us try to 'bend' the pointer a little by inputting the following:

```
poke 1104,peek(1104)+1600 <Return>
```

The display jumps down and can be checked using any window. The operating system is misled by the false pointer address and assumes the wrong memory area for the video display! Let's put everything back to avoid other unpleasant effects.

```
poke 1104,peek(1104)-1600 <Return>
```

This statement line restores the screen to its original state. However, there are many pointers that cannot be 'bent' as easily as the video display pointer. These vectors represent jump addresses (pointing to machine language routines) to which the system will branch. An important use of these jump addresses are the interrupt vectors. Interrupts are program interruptions that occur in order to allow the processor to perform needed internal tasks.

If a pointer is directed to an address that does not contain a machine language program, the ST crashes with the standard cherry bomb farewell. Since these interrupts usually occur many times a second, the crash is almost immediate. For this reason, leave these memory locations alone!

Let us select a harmless pointer as an example. This pointer is delivered by BASIC on demand and provides the position of a variable in storage. This is the VARPTR() function. This address can be used to manipulate text.

```
10    rem *** VARPTR()-Demonstration 2.1.4***
20    a$ = "Tricks":rem Text variable defined
30    a = varptr(a$) :rem Determine location
40    print a$              : rem Before .....
50    poke a,80 : print a$:rem and after !
```

"Tricks" is changed to "Picks" which occurs without direct manipulation of the variable A$. Through the use of vectors, the application possibilities of PEEK and POKE commands are considerably extended.

Here is another small program which permits you to POKE around in memory. After you input the memory address (0 stops the program) it displays the current contents of this word and asks for a new value. If you want to leave it unchanged, press <Return> and the next address will be investigated.

```
10  rem *** PEEK and POKE - Application 2.1.4***
20  input "Address";a          : rem input
30  if a=0 then end            : rem stop on 0
40  print peek(a); " => ";     : rem old value
50  input x$                   : rem new value
60  if Len(x$) = 0 then 20     : rem no change
70  poke a,val(x$) : goto 20 : rem loop
```

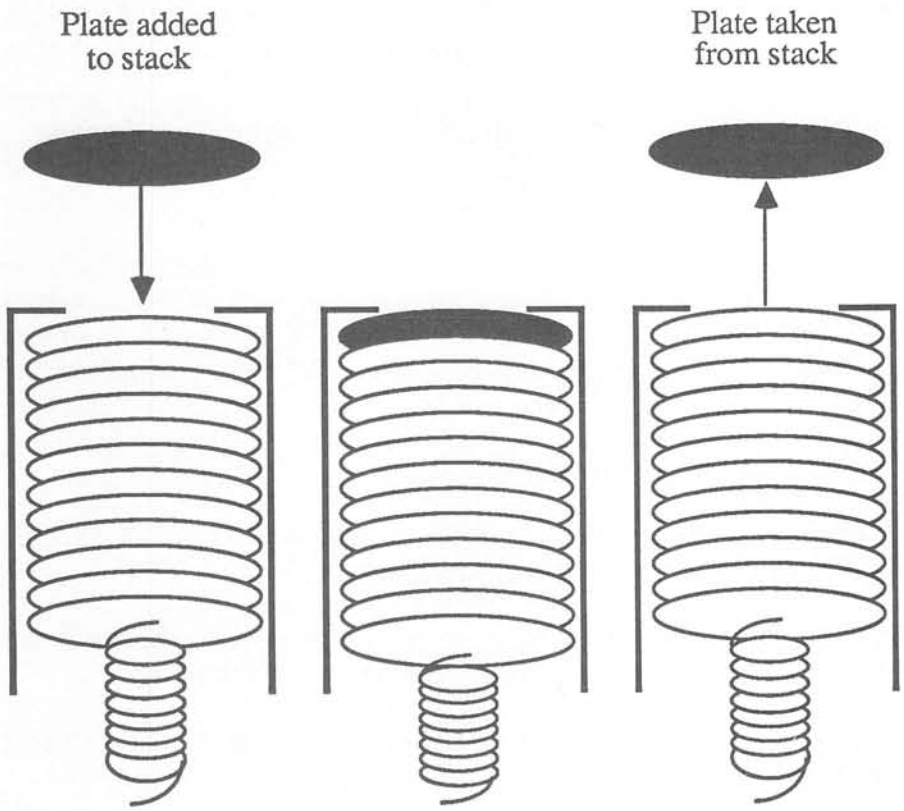
This program should be saved to disk, since you'll probably use it often to POKE around.

2.1.5 Stacks

A *stack* is a special area of memory used to temporarily store data.

The 68000 processor in the ST has two types of stacks: one is reserved for the operating system and the other for the user's application programs.

A stack in a computer works like the familiar spring-loaded plates in a cafeteria. When you place a clean plate onto the plate dispenser, any other plates are pushed down compressing the spring. When you want a plate, you remove the top plate and the remaining plates are popped up by the spring.



The stack works on a LIFO basis. This acronym stands for last-in-first-out. The last plate stored is the first plate retrieved. In the ST, the last data stored on the stack is the first data retrieved from it.

One of the many uses of a stack is to temporarily store addresses. For example, when the ST encounters a BASIC `GOSUB` statement it stores the address of this statement on the stack. Later, when it encounters the corresponding `RETURN` statement it retrieves this address from the stack and continues with the statement following the original `GOSUB`.

Stacks thus allow us to temporarily store small amounts of data not needed at the moment. For larger amounts of data you may consider other storage alternatives, for example floppy disks.

2.2 Disk storage

Floppy disk storage is the most common method of storing large amounts of data. The ST uses a 3.5 inch diskette. These are more expensive than the more common 5.25 inch diskettes, but are more durable and easier to handle. With either size, the storage technology is basically the same.

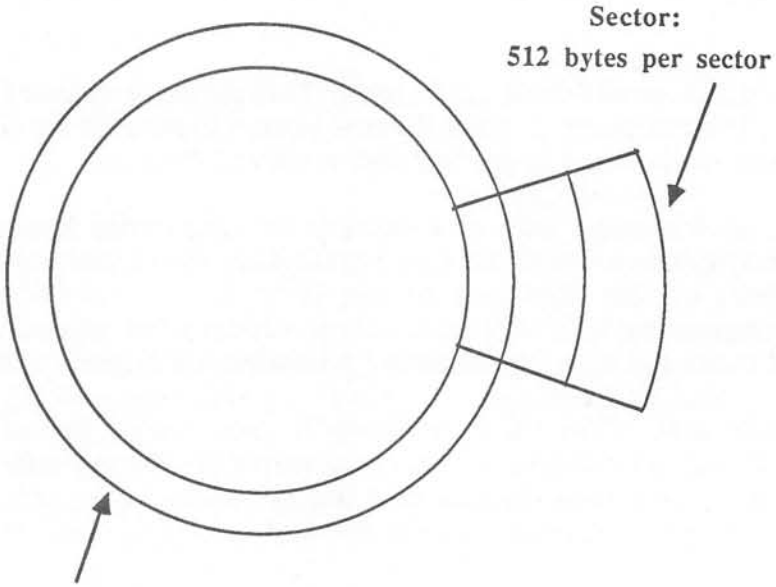
If you slide the metal safety cover on the 3.5 inch diskette, you'll see a dark brown 3.5 inch diameter "disc." It has a magnetic coating similar to that on audio cassettes. This "disc" turns inside the housing at a constant speed, where the disk drive's read/write head can transfer electrical impulses.

There are two varieties of floppy drives:

The SF354 is a single-sided drive with one read/write head that reads and records data on the bottom side of the diskette.

The SF314 is a double-sided drive with two read/write heads that read and record data on both the top and bottom sides of the diskette.

The surface of a diskette is electronically divided into tracks and sectors, illustrated in the following figure:



A track is like a "groove" in a phonograph record. While the diskette is spinning, the read/write head can remain stationary to access any information within that track. To access information in a different track, you'd have to move the read/write head to a different position.

A sector is a "slice" of a single track. Each sector represents a chunk of data that is transferred to/from the computer by the read/write head at one time.

A SF354 single sided drive has 80 tracks. Each track is made up of 9 sectors. Each sector stores 512 bytes of data. The resulting storage capacity is 80*9*512 = 368640 bytes.

A SF314 double sided disk drive has 160 tracks. Tracks 1 to 80 are on the bottom side of the diskette while tracks 81 to 160 are on the top side of the diskette. Again, each track is made up of 9 sectors and each sector stores 512 bytes. This makes the storage capacity 160*9*512 = 737,280 bytes.

In general, there are three major types of files: program files, data files and picture files.

2.2.1 Program Files

By itself, the ST can't do very much useful work or entertaining. We can change the entire complexion and personality of the ST with different programs.

Programs may be stored on cartridge or on diskette. Since most programs are available only on diskette, we'll ignore the cartridge variety.

From the last section we know that a diskette is organized in tracks and sectors. Finding a place to store programs or data on the diskette is not so easy. This is one of the functions of the operating system.

The operating system has built in functions for managing space on the diskette. When you load a program, the operating system uses these disk management routines to locate and activate the program from diskette.

When you save a program, these routines find space to transfer the program from the computer's memory to the diskette.

All of these activities take place transparently to the user.

2.2.2 Data Files

A second type of file is the data file.

You can store relatively large amounts of data on a diskette. This data is then easily exchanged by physically using the diskette on other ST computers.

The following is a short example that creates a data file on a diskette, allows you to write data to the data file and then rereads and redisplays the data file contents on the screen.

```
5     input "Enter a filename";f$: rem data, TEST.TXT
      2.2.1
10    input "(1) Input / (2) Output / (3) End";a
20    on a goto 50,100
30    end
40    rem *** Text Input ***
50    print "Input :  ('x'= termination)"
60    open "O", #1, f$, 128 : rem O for output
70    input a$
80    print#1,a$ : if a$="x" then close #1 :goto 10
90    goto 70
95    rem *** Text Output ***
100   print "Output :"
110   open"I", #1, f$, 128 : rem I for Input
120   input#1,a$ : if a$="x" then close #1 :goto 10
130   print a$ : goto 120
```

This program illustrates the OPEN, INPUT#and PRINT# commands. The direction of the data (to or from the disk) is specified by the OPEN command. With the PRINT# instruction you can transfer any desired text or numerical data from numerical variables and load them again using INPUT#.

2.2.3 Graphic Data Files

You can also store graphic "images" to diskette.

By using the BLOAD and BSAVE commands, you can load and save ranges of the ST's memory directly from/to the diskette.

For example, you can save the contents of video memory to a diskette, thereby saving it as a "snapshot."

Here's an example of how it's done.

```
10   gosub 100
15   fullw 2: clearw 2
20   gosub 200
30   end
100  rem *** Display.BAS  2.2.2***
110  defdbl a : a=1102
120  b=peek(a)
130  bsave "picture.dat", b, 32000
140  return
200  rem *** Load Picture ***
210  defdbl a : a=1102
220  b=peek(a)
230  bload "picture.dat",b
240  return
```

The two parts of the program are called as subroutines with GOSUB 100 or 200. The selected file name "PICTURE.DAT" can be changed. However, the file name can be at most 8 characters long and contain a three position extension (here .DAT) where the period does not count. Characters beyond this are ignored. Furthermore, you must be sure that the first character in the name is alphabetic (A - Z).

Chapter 3

Computer Mathematics

Computer Mathematics

Sometimes when you work with computers, especially in machine language or when using PEEK and POKE, you have to use a different number system. To make sure that we all are familiar, here's some background material on number systems. You can skip this section if you're already familiar with conversions.

Addresses in the material presented so far are expressed in *hexadecimal* or base sixteen. Normally a hexadecimal number is prefixed by a $ or suffixed with the letter H.

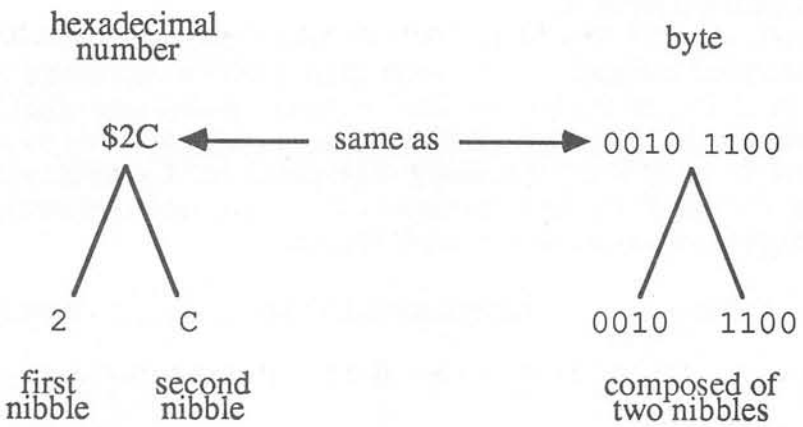
In our decimal number system each digit position represents a values from 0 to 9. But in the hexadecimal number system each digit position represents a value from 0 to 15. So we have to have a way to represent the values from 10 to 15 in a single digit position., Therefore we use the letters A through F to represent these values. The table below shows the corresponding values in each number system.

Decimal Value	Hexadecimal Value	Binary Value
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

By using the hexadecimal number system, large values can be represented more succinctly than in the decimal system. For example the value 16,515,072 decimal corresponds to $FC0000 in hexadecimal.

Another advantage is that the hexadecimal number system corresponds more closely to the division of words in computer memory. A single byte can assume a value between 0 and 255, or 256 different conditions. Using the hexadecimal system, each of the 256 values can be described with just two digits. For example, the value 255 is represented as $FF.

To more clearly see the "bit configuration" of the byte, we often divide the hexadecimal number into two *nibbles*. Thus the first hexadecimal digit is the first nibble and the second hexadecimal digit is the second nibble, as below:



As you might guess, the nibble is actually a half-byte whose binary value is equivalent to the hexadecimal digit. For some purposes, it is easier to visualize the value of a byte as two nibbles.

The 68000 processor uses neither the decimal nor the hexadecimal number systems. Since it is an electronic device it distinguishes between two different conditions or *states*, for example 0 volts or +5 volts.

On paper we represent these conditions using zeros and ones. This number system is called *binary*.

The decimal number 100 is represented in binary form as 01100100. Each digit position represents a power of two. The rightmost **bi**nary digi**t** (bit) position has a value $2^0=1$, the next $2^1=2$, the $2^2=4$ etc.

bit position	7	6	5	4	3	2	1	0
power of 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
decimal value	128	64	32	16	8	4	2	1

The binary value 01100100 thus corresponds to the decimal value 100 as follows:

$$
\begin{aligned}
& 0*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 \\
= \; & 0 + 64 + 32 + 0 + 0 + 4 + 0 + 0 \\
= \; & 100
\end{aligned}
$$

Representing numbers in the binary system takes time and a lot of writing space.

3.1 Number System Conversion

Conversions between the different number systems is rather cumbersome. But wait! These kind of calculations are what a computer is made for! Let's write a program in BASIC that can do it. The program listed below is just what we need. We had to perform a few tricks, since the ST BASIC loses some accuracy. Furthermore, it is restricted by the HEX$ function, which can only be used with a direct numerical value—for example, PRINT HEX$(100)—or with an integer value like PRINT HEX$(X%). It can output hexadecimal numbers only up to $FFFF.

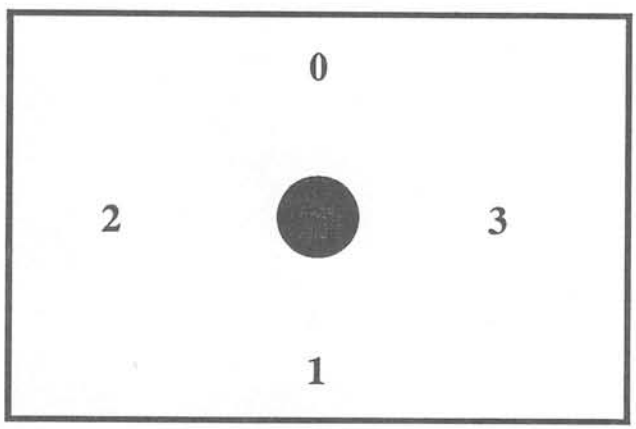
```
10    defdbl e,d  : rem CONVERT.BAS 3.1
20    defint n
21    clearw 2: print "Select one: "
22    print"1) Decimal to hex 3)Decimal to Binary"
23    print"2) Hex to decimal 4)Binary to decimal"
24    print : input a
25    on a goto 30,110,170,210
26    goto 21
30    z$="0123456789ABCDEF"
40    input "Decimal number ";d: rem Decimal to Hex
50    for i = 5 to 0 step -1 :e = 16^i
60    n = .1 + d / e : d = d - n * e
70    print mid$(z$,n+1,1);
80    next i :? :? h$ :end
110   z$= "0123456789ABCDEF"
120   input "Hex Number ";h$  : rem Hex to Decimal
130   d=0 : for i = 1 to len(h$)
140   e = 16^ (len(h$)-i)
150   d=d + e * (instr(1,z$,mid$(h$,i,1))-1)
160   next i : print int(d) : end
170   input "Decimal number ";d:rem Decimal to Binary
180   print d;" => "; : for i = 15 to 0 step -1
190   if (d and 2^i) then ? "1"; else print "0"
200   next i : print : end
210   input "Binary Number ";b$:rem Binary to Decimal
220   print b$;"="; : d = 0
230   for i = 1 to len(b$) : e = len(b$)-i
240   d= d - (mid$(b$,i,1) = "1") * 2^e
250   next i : print int(d) : end
```

3.2 Bit Evaluation

Binary representation is important to manipulate the values in memory. Often certain functions are dependent on one bit—for example, the joystick position. The number that is returned from reading the joystick represents the condition of the four switches that are activated by the stick movement. Bits 0 and 1 represent the vertical movement of the joystick. Bits 2 and 3 are for the horizontal direction. The following small program demonstrates this:

```
10    rem ***Joystick - Evaluation 3.2***
20    out 4,22  : rem Instruction to Keyboard
30    defseg = 1
40    js = peek(3591) : rem Get Result
50    print js, : rem Output Total
60    Y = Y + (js and 2)/2 - (js and 1)
70    X = X + (js and 8)/8 - (js and 4)/4
80    print x,y : rem Output Coordinates
90    goto 20   : rem Infinite Loop
```

This program reads the joystick and changes the position given by X and Y in relation to its position. The corresponding bits are tested with the AND command. These AND relations are closely related to the BASIC function OR, NOT and XOR. The comparisons are made in binary by comparing the two values bit for bit. The results of the logical comparisons are discussed in the next section.



Joystick Switches

3.3 Logical Operators

There are four main logical operators: AND, OR, XOR, and NOT. These logical operators serve to manipulate and test bit combinations.

Table: Logical Operators

AND

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

The bit in the result is set only when both bits tested are 1.

OR

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

The bit in the result is set always when one of the tested bits is 1.

XOR

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

The bit in the result is always a 1 when when the tested bits are different.

NOT

```
NOT 0 = 1
NOT 1 = 0
```

The sample program of the joystick evaluation tested the value of JS with the AND condition. If this value is equal to 3 and if bit 1 should be tested, the operation JS AND 1 proceeds in the following manner:

```
JS    = 3,        corresponds to binary 0011
Testbyte = 1,                     binary 0001

                                          ____
Result            3 AND 1:                0001
```

The result never equals 0 when the tested bit is a 1. Using this method we can test the joystick with four such comparisons.

Another application of the logical operators is the direct manipulation of a bit. If there is a data byte in which a certain bit should be changed to 0, the AND instruction can be used.

Example:

```
Data byte is    00110011 to change bit 4
         AND    11101111 this results in

Bit changed     --------
                00100011
```

This process is called *masking*. The 11101111 word represents the mask. A similar method is used to set a bit to 1. The OR instruction is used here:

Example:

```
Data byte is    00110011 set bit 5 to 1
so we      OR   00001000

Bit changed     --------
                00111011
```

The XOR instruction is an interesting function. It permits bits to be set to 0 or 1. If the same operation is performed several times the bit alternately changes from 1 to 0.

Example:

```
Data byte is    00110011 bit 0 is changed
         XOR    00000001

                --------
results in      00110010 and again
         XOR    00000001 again results in

                --------
                00110011 the original value!
```

The NOT operation is similar to the XOR command. NOT inverts all bits in a data byte. This produces a value that corresponds to the decimal 255 minus the original byte.

Example:

```
Data Byte is    00110011 = 51
        NOT     00110011 results in

                11001100 = 204 + 51 = 255
```

In addition, there are two special operators which are supported by ST BASIC—shift left and shift right. You can shift a data word left or right. The effect is to double the value of the word, or to cut it in half. An example:

```
Data byte           00110011 = 51
left shifted        01100110 = 102, also 51*2
right shifted       00011001 = 25, also 51/2 (integer)
```

The shifted bit position is filled with zeros after the operation. In short this means that n shifts to the left correspond to a multiplication of 2^n and n shifts to the right means a division by 2^n.

When using BCD (Binary Coded Decimal), each byte of memory is divided into two nibbles. But these nibbles may only take on values from 0 to 9. The 68000 processor operates on these values as decimal values, not hexadecimal. Thus a 16 bit word can represent numbers from 0 to 9999.

The advantage of BCD lies in the simple representation and processing of decimal numbers. The conversion of a BCD coded decimal number is as simple as extracting the hexadecimal value from a normal binary number. You can convert each nibble and obtain the coded value directly. An example of the BCD coding is found in the time program in the chapter on the intelligent keyboard. The real time and date are stored in memory in BCD.

This is in contrast to the hexadecimal number system where values from 0 to 65,535 may be represented in 16 bits.

Chapter 4

Operating Systems

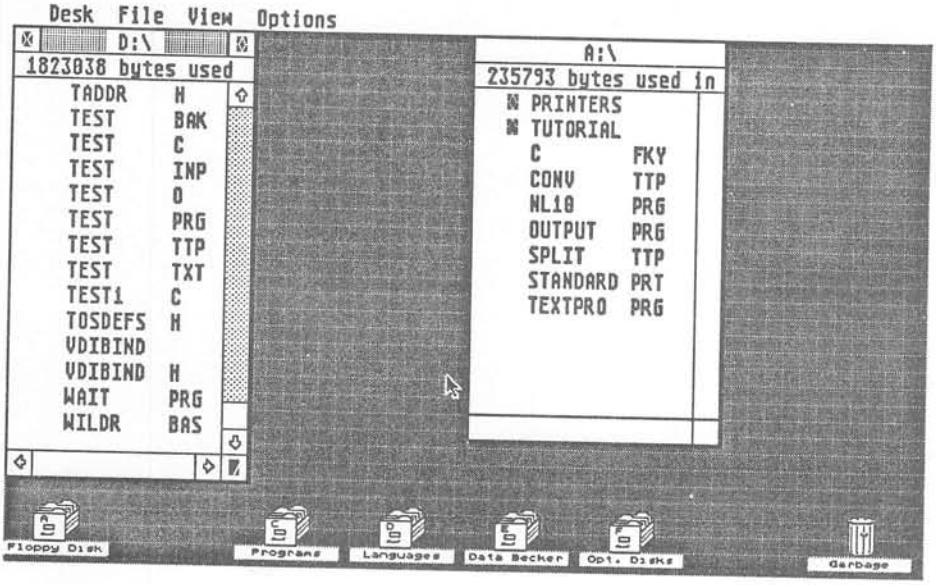
The Operating System

A computer is really a remarkably "dumb" machine. Unless it is told what to do, it can't perform any useful work.

The *operating system* is a set of instructions (or a program) which tells the computer what to do. In earlier computers the operating systems were *command oriented*—you typed in a command and the computer responded.

The ST, on the other hand, has a different type of operating system. It is *icon oriented*—you point the mouse at a pictorial representation and the computer responds.

The ST's operating system is made up of several parts. We'll talk about them now.



4.1 The Tramiel Operating System

The ST's operating system was named after the "Father" of this remarkable computer—Jack Tramiel. TOS maintains the diskette and peripheral controls established with its predecessor operating system CP/M, but TOS can do considerably more and at a faster speed. TOS has borrowed some functions and behavior from CP/M, but additional capabilities have been built in as well. For example, the ST has a hierarchical directory structure which is not found in the earlier CP/M. This makes it possible for an ST user to build subdirectories (folders).

In the newest STs, the TOS is built into the ROMs on the motherboard and is activated automatically when the computer is turned on.

Earlier versions loaded TOS from disk into the memory area beginning at $500 up through $32000. In addition, TOS contains GEM, the graphics user interface (Graphics Environment Manager) and the resource parameters to create the *menu* and *alarm* windows.

When TOS is activated, the BIOS starts into action and places the ST into a startup condition. It searches the diskette for programs with the `.ACC` suffix designators. If any are found, these accessories are loaded into memory and are then available for use when accessed form the `DESK` menu.

4.1.1 The BIOS

One important part of the TOS is its Basic Input/Output System (BIOS). The BIOS is the interface between the physical hardware and the software. It performs rudimentary and frequently used operations using the input and output devices. A programmer does not have to concern himself with the details of accessing a peripheral. Instead he requests the services of the BIOS.

The BIOS in the ST is enhanced compared to its predecessors. It can handle operations to the MIDI interface, for example. There are more than 50 "services" or commands which are built into the BIOS and XBIOS (extended BIOS). These services are available to the BASIC

programmer by PEEKing and POKEing. Several examples are presented in later chapters.

4.1.2 System Variables

At startup, the BIOS initializes another area of memory—the table of system variables. This table contains many pointers or vectors which point to interrupt routines—machine language programs which are called to handle program interruptions. If these vectors are inadvertently changed, the system will probably crash.

The table is located beginning at $400 through $4FF (1024 - 1279 decimal) and contains the following:

$400	Event timer of GEM. Handles periodic tasks for GEM
$404	Critical error handler
$408	GEM vector for ending a program
$40C	Space for 5 additional GEM vectors which are not used at this time.
$420	Contains a flag which indicates a successful cold start
$424	Memory configuration (= 4 with 512K, 5 with 1 Mbyte)
$426	Again a flag which causes a cold start on Reset
$42A	Pointer to the cold start routine
$42E	Pointer to the end of RAM ($80000 with 512K and $F0000 with 1 Mbyte)
$432	Pointer to start of working memory

$436	Pointer to end of working memory
$43A	Another flag indicating a successful cold start
$43E	DMA flag must be <>0!
$440	Floppy speed set (3)
$442	System-Timer in milliseconds (20 for 50 Hz)
$444	Floppy comparison flag. When <>0, the every write is tested with a read. If 0, then no test (writes faster)
$446	Unit number from which the system was loaded
$448	PAL/NTSC Flag <>0: PAL(European systems) =0: NTSC (American systems)
$448	Display resolution set
$44C	Actual display resolution (0-2)
$44E	Pointer to display start
$452	VBI Flag. Should be a 1 (VBI = Vertical Blank Interrupt
$454	VBI routine number (8)
$456	Pointer to VBI pointer table
$45A	Pointer to new color table (when new)
$45E	Pointer to new screen memory (when new)
$462	VBI counter

$46A	Completed VBI counters
$46E	Pointer to routine at monitor change (cold start)
$472-$481	Hard disk parameter (0 when not present)
$482	When <>0, COMMAND.PRG was loaded
$484	Keyboard status (bit 0: bell on/off; bit 1: key repeat on/off; bit 2: key click on/off
$48E	Working memory limit (do not change!)
$4A2	Pointer to BIOS register memory
$4A6	Number of disk drives connected
$4AE	Pointer to condition of computer memory
$4BE	Pointer to data sector intermediate memory
$4B8	Pointer to the directory buffer
$4BC	200 Hz counter
$4C4	Is 3 when floppies are connected
$4C6	Pointer to 1K disk buffer
$4EE	Hardcopy flag (when 0 then print display)
$4F2	Pointer to start of operating system
$4F6	Pointer to graphic or text segment
$4FA	Pointer to end of operating system
$4FE	Pointer to AES (Application Environment Services) text segment

4.1.3 Talking to the TOS

Most of us are familiar with only the GEM desktop. Its characteristics are its familiar icon-oriented screen and drop-down menus.

In addition, you can talk directly to TOS. Programs that bypass GEM and talk directly to TOS have the extension `.TOS`.

One way to talk to TOS without writing a program is to use the VT52 emulator found in the `DESK` submenu. Here the keys have a different "meaning." A cursor key function is a sequence of key presses—the first being the <ESC> key:

<Esc> A	Cursor up
<Esc> B	Cursor down
<Esc> C	Cursor right
<Esc> D	Cursor left
<Esc> E	Clear display and cursor to upper left
<Esc> H	Cursor to left upper corner
<Esc> I	Cursor up, scroll if required
<Esc> J	Clear display starting at cursor
<Esc> K	Clear line starting at cursor
<Esc> L	Add line
<Esc> M	Delete line
<Esc> Y	(y-32)(x-32) Bring cursor into X/Y position
<Esc> b	Color Select color of writing (0-16 with color)
<Esc> c	Color Select color of background (0-16)

<Esc> d	Delete display up to cursor
<Esc> e	Display cursor
<Esc> f	Delete cursor
<Esc> j	Store cursor position
<Esc> k	Set cursor to stored position
<Esc> l	Delete line
<Esc> o	Delete line up to cursor position
<Esc> p	Start reverse video
<Esc> q	Stop reverse video
<Esc> v	Wraparound on
<Esc> w	Wraparound off (cursor stops at right margin)

These key sequences can also be used from a BASIC program. It is possible to erase the output window border and use the entire screen for the program. But be careful! An erased menu line remains erased even though it may still function. Therefore you have to redraw it after returning to the BASIC level.

The following short program demonstrates talking to TOS:

```
10    rem *** TOS Level Demo 4.1.3***
20    x = inp(2): rem Get Key
30    if x = 187 then end : rem F1 causes
      termination
40    out 2,x    : rem Output to TOS Display
50    goto 20    : rem Infinite Loop
```

This trick can be used in a BASIC program to get two cursors which are independent of each other. To control the TOS cursor from the BASIC program, use this program:

```
10   rem ***Output to TOS Cursor 4.1.3***
20   esc$=chr$(27)          : rem Define Escape
30   aus$ = esc$+"e"+esc$+"H"+esc$+"B"+
     esc$+"p":rem home, cursor down, reverse
40   aus$ = aus$+"Hello!" : rem Text
50   for i=1 to Len(aus$): rem Output Loop
60   out 2,asc(mid$(aus$,i,1)):rem Pass
     Character
70   next i              : rem until done
```

The use of the AUS$ variables can also be done differently. The output loop passes the entire text, including the control character, to TOS. All control characters shown above may be used.

4.2 GEM

GEM is the Graphics Environment Manager. It was developed by Digital Research as a friendly user interface. It resembles the interface of the Apple Macintosh® which first demonstrated this type of graphic oriented interface. Here the usual method of entering commands through the keyboard is replaced by using icons and menus. The advantage lies in ease of comprehension of the computer commands—most of the functions are self explanatory.

It is interesting to use the capabilities of the GEM in your own programs. GEM has all the functions required to create graphics and to use menus. But before using it, you should understand how it works and how it is structured.

GEM actually consists of two parts: the VDI (Virtual Device Interface) and the AES (Application Environment System). Although they have different tasks, their services are accessed similarly.

The VDI is responsible for drawing graphics. It is made of the GDOS which performs device independent graphic functions, and the GIOS which performs the device specific graphic functions. By implementing the GEM in this way, it's possible to quickly add a special "driver" to the GIOS for a new graphic peripheral.

The AES manages the user interface. It is responsible for handling drop-down menus, icons, windows, etc. The AES frequently call upon the VDI to draw its graphic displays.

4.2.1 GEM Programming from BASIC

Both the VDI and AES were designed for easy use from the C language and machine language. The VDI and AES routines are contained in a *library*. To use the routines, you must pass parameters to them.

The C and machine language programmer can easily access the routines. But the BASIC programmer has a little more work to do to use the features of GEM. Luckily ST BASIC has two built-in commands to

access the VDI and AES. These commands are GEMSYS and VDISYS. The GEMSYS command talks only to the AES, while the VDISYS command accesses the VDI.

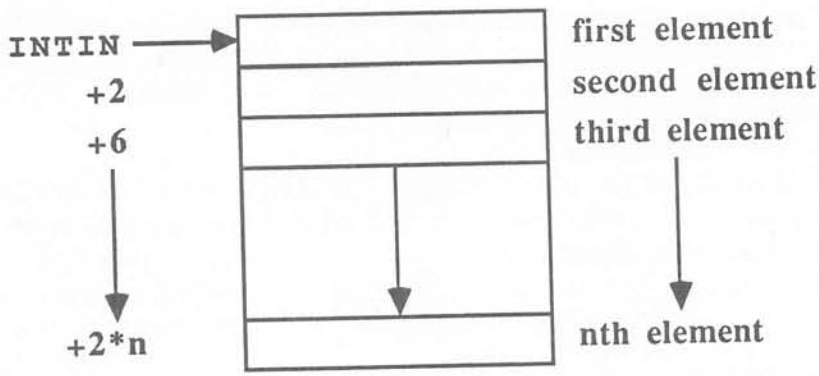
For passing parameters to or from the VDI and AES, ST BASIC reserves several variables: INTIN, INTOUT, PTSIN, PTSOUT and CONTRL.

By displaying the values of these variables, you can determine the starting memory addresses of the various parameter arrays, e.g. PRINT INTIN. A reference to the variable name in a PEEK or POKE statement references its address. This is a convenient way to access the elements of the array.

Each array element is 16 bits (or 2-bytes) wide. You can display the contents of the third element of the INTIN array with the following:

```
PRINT PEEK(INTIN + 6)
```

Since each element is 2-bytes wide, the third element is found by multiplying the element number by two.



The array named INTIN is for passing input parameters to GEM. The output parameters from GEM to the program are passed in the array named INTOUT. The arrays PTSIN and PTSOUT are used to pass coordinates for graphic functions. And the CONTRL array is used to specify the desired function.

Here's a breakdown of the CONTRL array:

CONTRL(0) Function code

CONTRL(1) Number of entries in the PTSIN array

CONTRL(2) Number of entries in PTSOUT array

CONTRL(3) Number of entries in the INTIN array

CONTRL(4) Number of entries in the INTOUT array

CONTRL(5) Function ID for subroutines

CONTRL(6) Unit number (handle)

CONTRL(7-n) Function dependent values

The second, fourth, and sixth to the nth elements are the output parameters which are returned by the routine. The others must be specified for every function call. The entry for CONTRL(6) is a number which GEM requires for the device identifier or unit to be accessed. Since the screen is active and represents the actual unit, this ID is not transmitted. All functions are performed on the screen.

4.2.2 Getting Input from the Mouse

Let's break away from concepts and get down to practice. This short program determines the mouse's screen position and the state of its button. The VDI has a function exactly for this purpose:

```
10    rem *** Get Mouse Position 4.2.2***
20    poke contrl,124     : rem Function Code
30    poke contrl,+2,0    : rem Number of Parameters
70    vdisys 0             : rem and Execution
80    x = peek(ptsout)    : rem X Position
90    y = peek(ptsout+2): rem Y Position
100   key = peek(intout): rem Key activated
110   print x,y,key
120   goto 20
```

For all calls to the GEM, you must specify a function code in element CONTRL(0). The number of parameters passed is specified in CONTRL(1). In this example there are no parameters so we POKE a value of zero. Next the call to VDI is performed. The value following VDISYS is a dummy argument.

The VDI always returns the output values in INTOUT or PTSOUT. In rare cases you may obtain values in the CONTRL array, but the two OUT fields are more important. In the previous example, the X and Y position of the mouse pointer in PTSOUT(0) and PTSOUT(1) are returned. The value in INTOUT(0) is zero when the mouse key has not been pressed. This information can be used in the program. Lines 110 and 120 are only for demonstration and may be omitted.

Now let's change the mouse pointer to a different shape. Perhaps you want to push a little man across the screen? You can do this through the VDI. For this application the following program passes a lot of parameters. Enter the following program.

4.2.3 Changing the Mouse Form

```
10    rem *** Set Mouse form 4.2.3 ***
20    poke contrl,111    : rem Function Code
30    poke contrl+6,37   : rem Number of Parameters
40    poke intin,3             : rem Action Point X
50    poke intin+2,0           : rem Action Point Y
60    poke intin+6,0     : rem Color value of Mask
70    poke intin+8,1           : rem Data color value
80    for i=0 to 15 : read x$ : rem Mask/Cursor
82    x=0 : for j=1 to 16      : rem Change
84    x= x-(mid$(x$,j,1)<>" ")*2^(16-j)
86    next j
90    poke intin+10+i*2,x      : rem Set Mask
92    x=0 : for j=1 to 16      : rem Conversion
94    x= x-(mid$(x$,j,1)="*") *2^(16-j)
96    next j
98    poke intin+42+i*2,x      : rem Set Cursor
100  next i
110   vdisys 0     : rem and execution
```

```
120   end
130   rem +++ Masks and cursor data +++
140   data "      ...          "
150   data "  ..  .***.        "
160   data ".**..*****.       "
170   data ".**..****.        "
180   data "  .**.**.          "
190   data "   .*******.       "
200   data "    .*****.**.     "
210   data "    .*****..*.     "
220   data "    .*****.**.     "
230   data "    .**.**....     "
240   data "    .**.**.        "
250   data "    .**.**.        "
260   data "  ...**.**...      "
270   data "  .****.****.      "
280   data "  ..........       "
290   data "                   "
```

This program probably needs some explanation. In line 20, the function code is specified as before. Next the number of parameters are specified. This function, whose name is `vsc_read`, reads 37 words from the `INTIN` array. `INTIN(0)` and `INTIN(1)` contain the X and Y coordinates of the action points of the mouse pointer. This point, for example, activates the drop-down menus. The action point in the above example was set at coordinates 0,3 in other words the right hand. The following two values in `INTIN(3)` and `INTIN(4)` contain the the color values of the cursor. You might wonder why two values are needed for this. The reason is simple. Assume the cursor is black and is dragged onto a black surface. It becomes invisible and therefore useless for exact positioning. Now the *mask* makes its appearance. This mask lies almost below the cursor and is slightly larger than the cursor. Its color value in the above example in normal desktop operation is 0, which means it appears white. The effect is that on a black surface you cannot see the entire cursor, only its outline. This outline is the white mask.

Starting in line 80, the program passes the data for the cursor and mouse form. Here again the binary-decimal conversion is used which was presented in section 2.1. For this application, the conversion routine was changed. For the computation of cursor data, which provides the binary pattern, the * is recognized as 1. The mask on the other hand is represented by periods since it is larger than the cursor itself. Here a

space is assumed to be a logical 0 and everything else between lines 82 to 90 represents a 1. Through this trick both of the tables for cursor and for the mask are saved.

The conversion program in lines 80 to 100 creates, from a combined table, a single table and passes it directly to the INTIN field. The data for the mouse form in INTIN(5-20) and the cursor data table in INTIN(21-36) are passed to the function. If you select any of the FILE functions of the Main Menu, GEM changes the mouse form to a "busy bee" and then back to the original arrow.

To use this in a program, it's not necessary to pass the binary form of the data table. We recommend that after you design the final mouse pointer form, you write the two data tables as decimal numbers. The program would look as follows from line 80 onward.

```
80    for i=0 to 31              : rem Start Loop
90    read x                     : rem Read value
100   poke intin+10+i*2,x        : rem and set
110   next i
120   vdisys 0                   : rem and execution
130   end                 : rem finished, maybe RETURN
140   rem ---Mask Data---
150   data 384, 2016, 8184, 32766,65535, 62415, 62415
160   data 62415, 960, 960, 960, 960, 960, 960, 0, 0
170   rem---Cursor Data---
180   data 0,384, 2016,18184, 29070, 24966, 384, 384
190   data 384, 384, 384, 384, 384, 0, 0, 0,
```

This example draws a vertical arrow instead of the little man from the previous program. The action point is now located at coordinates 8,0. If you prefer the little man form better, you can determine the values for the DATA statements from Section 2.1 by using the binary-decimal conversion program.

A final note about the sample program. There are practically no limits to the cursor form. Be careful to stay within the 16 x 16 pixel format or you may end up creating an unmanageable form.

There is also another way to change the mouse form. There are 8 pre-defined mouse forms built into GEM. For example, the "busy bee" is one of these. You can select this or one of the other seven with a call to the

AES as in the next program. We'll talk more about this shortly when we describe menu and window programming.

```
10   rem***Set Mouse form 4.2.3***
11   defdbl b,d : b=gb
12   cn=peek(b)
14   ii=peek(b+8)
20   d=peek(b+16)    : rem Create Pointer
50   poke cn,78           : rem Command
60   poke cn+2,1
70   poke cn+4,1
80   poke cn+6,1
90   poke cn+8,0
100  poke ii,4            : rem Choose Form
110  poke d,257           : rem Plug in Form
120  gemsys 78            : rem execute
```

The value set in line 100 represents the form. You can use any of these in place of the 4:

0 Arrow
1 Cursor
2 Bee
3 Hand with Index Finger
4 Flat Hand
5 Thin crosshair
6 Thick crosshair
7 Crosshair as frame

When you select a FILE function, the arrow form or busy bee form is reselected.

4.2.4 Changing the Font

GEM has the capability to represent text in various ways. In addition to the normal font, you can add the underlined characters, bold printing, outlined characters and shaded characters which are familiar to you from the BASIC editor. To use these variations, we have to call on the VDI again. A program to change the appearance of the font appears as follows:

```
10    rem ***Change Font 4.2.4***
20    poke contrl ,106  : rem Function Code
30    poke contrl+2,0   : rem Parameter Number
40    poke contrl+6,1
50    poke intin  ,1+2        : rem Font
60    vdisys 0                :rem and execute
70    print "What kind of Text !" : rem Test Text
80    poke contrl  ,106 : rem Loop
90    poke contrl+2,0
100   poke contrl+6,1
110   poke intin   ,0   : rem Normal font
120   vdisys 0          : rem used again
```

In this program the font is changed to bold and shaded style. After displaying a sample line, it reverts to the normal font to avoid potential problems with subsequent BASIC commands. Lines 10 to 40 are similar to the previous example. The difference lies in line 50 where the desired font is specified. This method makes font selection easy. The text modes can be selected by combining the bits which represent the desired graphic characteristics. These are represented as follows:

Bit	Value	Font
0	1	Fat
1	2	Shaded
2	4	Italics
3	8	Underlined
4	16	Outlined

To use a shaded, italics font with underlining, then line 50 should have the value 2 + 4 + 8, or 14. Up to 32 styles can be selected by using

various combinations. Some of the styles such as the outlined, cursive font (16 + 4 = 20) are illegible, unless you have a lot of imagination.

4.2.5 Graphic Text

We still haven't exhausted the capabilities of the VDI. We can directly format text output and display it at any location on the screen. Thus we can display text outside the output window of BASIC and even within the menu lines. Here's how to do it:

```
10    rem *** Graphic Text - VDISYS Demo 4.25***
15    text$ = "Sample Text"
20    poke contrl,11          : rem Function Code
30    poke contrl+2,3         : rem Parameter Number
32    poke contrl+6,len(text$)+2
34    poke contrl+10,10 : rem Function Recognition
40    poke intin+2,1          : rem Word Expansion
50    poke ptsin  ,50         : rem X Coordinate
60    poke ptsin+2,60         : rem Y Coordinate
70    poke ptsin+4,150        : rem X-Text length
80    for i=1 to len(text$)   : rem ASCII Code
90    poke intin+2+i*2,asc(mid$(text$,i,1)) : rem Set
100   next i
110   vdisys 0          : rem execute
```

This program displays the text from the variable TEXT$ on the screen at coordinates 50,50 and stretches the text to the point where it reaches the width of 40 characters. The 3 which is passed to the control array in line 30, gives the number of parameters to be passed in PTSIN. The first two are the coordinates of the first character of text, where 0,0 indicates left on top. The X-Text length is the total width which the text shall occupy. The text is stretched by adding space between the characters. To write the text unexpanded to the location X, the expansion mode can be switched off by POKEing a value of zero in line 40.

The program also performs some text processing. The VDI must know how many characters to display. This number comes from line 32 through the parameter which it should read from the INTIN array. The function LEN(TEXT$) outputs the length of the text in this variable. The +2

stands for INTIN(0) and INTIN(1). The text is written character by character into the INTIN field.

To store a character in memory, you must write its ASCII value. ASCII stands for "American Standard Code for Information Interchange" and is a standard code used in most computers and printers. The function ASC("A") provides in a BASIC program the ASCII value of A, which is 65. Every character has its own value, so that in our example any desired text can be set into the variable TEXT$. The character to be converted is selected from the string using the MID$ function.

However, we want to postpone text processing until later (section 7.4) Let's continue our discussion of the VDI. The interesting VDI commands are explained in the sample program in chapter 6.1. For a complete overview of the VDI and AES commands, whose applications are mostly limited to C and machine language programming, see the *Atari ST GEM Programmer's Reference* from Abacus Software.

An interesting block of memory in which some parameters of BASIC are stored for use by GEM, is directly accessible through SYSTAB system constants. SYSTAB is a pointer which points to this parameter block. Through PEEK and POKE, some interesting effects can be achieved. Some of the parameters that can be reached are only suitable for reading since changing them could easily lead to a system crash. The following are the addresses and their significance:

SYSTAB	Graphic Resolution (1=high, 2=medium, 4=low)
SYSTAB+2	Editor action, font (see below)
SYSTAB+4	Edit window AES Access code
SYSTAB+6	List-Window AES Access code
SYSTAB+8	OUTPUT window AES Access code
SYSTAB+10	COMMAND window AES Access code
SYSTAB+2	Editor action, font (see below)
SYSTAB+12	EDIT Flag (0=Closed, 1=Open)
SYSTAB+14	LIST Flag (0=Closed, 1=Open)
SYSTAB+16	OUTPUT Flag (0=Closed, 1=Open)
SYSTAB+18	COMMAND Flag (0=Closed, 1=Open)
SYSTAB+20	Pointer to Graphic Buffer
SYSTAB+24	GEM-FLAG (0=normal, 1=out)

The graphic resolution is 1 with a monochrome monitor and 2 or 4 with color. A PEEK(SYSTAB) allows you to determine resolution. This is

sometimes necessary in graphics programs since you will have to adjust the maximum X/Y coordinates.

The editor action and font specify the way the currently edited line is displayed. Normally, it is displayed in a lightly shaded font style. With `POKE SYSTAB+2,14` you can change this to a Roman font.

The AES access codes of the individual windows of the BASIC workbench are the numbers which the AES has assigned to the windows. Using this number with an AES call you can change the desired window (for example enlarge, move, or make smaller).

The flags of the window conditions contain information on the condition of each window. You can look here to determine if a window exists. Be careful in changing the flag values. A `LIST` command can crash the system if the `LIST` flag is cleared.

At memory locations `SYSTAB+20` to `SYSTAB+23` is a 32 bit long word, the memory address of the graphic buffer. This buffer allows you to alter the `OUTPUT` window during size changes. The address of this memory area can be determined by accessing the long word which starts at `SYSTAB+20`.

And now to the GEM flag. If you set it to 1 with `POKE SYSTAB+24,1`, GEM is turned off. You won't notice anything immediately, but changes are no longer possible. No input is accepted from the keyboard either. The GEM flag can only be used within a program, but it must be reset to zero at the end of the program. The advantage of setting the flag in the first place lies in the time which GEM normally requires to perform windowing and to manage menus. These activities are suspended when GEM is turned off and the computer can devote more time to the processing of BASIC programs. The program runs faster, which during long computations can be advantageous. Disk access is permitted since it is handled by BIOS. Don't forget to turn GEM on again!

Chapter 5

The Desktop

The Desktop

5.1 Customizing the Desktop

Many users want to customize their ST's to satisfy their own preferences and work habits. Let's see how you can customize the ST for yourself.

Most of the icons that appear on the desktop are difficult to change using the accessories. For example, the disk drive icons can only contain capital letters if you try to name them with the `Install Disk Drive` option from the `Options` menu.

There is another way to change these icons, however. We must change the `DESKTOP.INF` file. This file contains all the information the ST needs to create the desktop. The ST uses this file each time you boot the computer.

Caution: You should only experiment on a back-up diskette because it is very easy to destroy this file. If you are using a hard disk drive you should disconnect it before experimenting with the desktop. You can load `DESKTOP.INF` into any text processor with an ASCII mode. Just be sure to set all the margins to 0. The file should look similar to this:

```
#a000000
#b001100
#c77700070007000700552005055522207705570750577055041123 06
#d
#E 9B 03
#W 00 00 0C 01 1D 16 08 A:\*.*@
#W 00 00 28 01 1F 17 00 @
#W 00 00 0E 09 2A 0B 00 @
#W 00 00 0F 0A 2A 0B 00 @
#M 00 02 00 FF A FLOPPY DISK@ @
#M 00 03 00 FF B FLOPPY DISK@ @
#T 00 07 02 FF TRASH CAN@ @
#F FF 04 @ *.*@
#D FF 01 @ *.*@
#G 03 FF *.PRG@ @
#F 03 04 *.TOS@ @
#P 03 04 *.TTP@ @
```

Line `#a` is for the RS232 configuration. The digits in this number are related to the settings of the RS232 configuration menu. The first digit is DUPLEX. It can be set to 0 for Full or 1 for Half duplex. The second digit is for the baud rate. It can be set to 0 for 9600 baud, 1 for 4800, 2 for 1200 or 3 for 300 baud. Digit number 3 is for the Parity. Set it to 0 for None, 1 for odd or 2 for Even parity. The fourth digit is for Bits/char where a 0 represents 8 bits/char, 1 is for 7, and so on down to 5 bits/character. Digit 5 is for the XON/XOFF and RTS/CTS settings see the diagram below. Digit 6 is for the Strip bit , set it to 0 to turn the strip bit on or set it to 1 to turn it off.

XON	RTS	digit 5
		0
*		1
	*	2
*	*	3

Line `#b` controls the printer configuration. The digits correspond directly to the `Install Printer` accessory menu. For example, the 4th digit corresponds to the Quality settings on the Install Printer menu. You can put only 1s and 0s in these digits. A 0 represents the choices in the first column on the menu such as DOT, B/W, 1280, etc. A 1 represents the second column.

Line `#c` contains the color palette values. These can also be set with the control panel. Each color is set by a group of 3 digits that correspond to the Red, Green and Blue values.

Line `#d` is not used at this time.

Line `#E` is concerned with how folders are viewed, i.e. as text or icons and how the files are sorted. This is easier to change with the `VIEW` menu. If you do wish to set these using the `DESKTOP.INF` file here is a table with all the settings:

Show as	Sort by	Set 1st value to
Icons	Name	1B
Icons	Date	3B
Icons	Size	5B
Icons	Type	7B
Text	Name	9B
Text	Date	BB
Text	Size	DB
Text	Type	FB

The #W lines control the windows. The first two values (in this case 00 00) are for the horizontal and vertical sliders. The zeros mean they are inactive. The next two values (0C 01) indicate the X and Y values of the window in units of characters. The two values following that (1D 16) contain the width and height of the window in character units. The information following this concerns the path for the disk directory. It's not wise to change this path information.

The #M lines control the disk icons. The first two values (00 02) are the position at which the icon will appear. The next value determines the shape of the icon.

00	file drawer
01	document
02	trash can
03	program
04	folder

The FF value doesn't seem to do anything. The floppy disk labels that follow can be changed using both upper- and lowercase letters.

The #T line is for the trash can. Its parameters are the same as those for the disk icons.

The #F line contains information for displaying folders in directories.

The rest of the lines are similar to the #F line except #D is for documents, #G for GEM files and #P for TOS Takes Parameters files.

After you make your changes save the file under the name DESKTOP.INF in ASCII format.

To use your customized desktop: first shut off your ST, put the disk containing the new DESKTOP.INF file in drive A and switch the computer on. Your new desktop will appear on the screen.

5.1 Setting the RS-232 Interface

You can only have 6 accessories on the desktop at a time. You may want to remove some of the accessories to make room for others or just to free up some memory. To remove an accessory, rename it with the extension .AC1. To re-install it as an accessory, rename it with the extension .ACC. When you reboot the system the computer will load all the files with the .ACC extension. Remember that you are only allowed 6 accessories at time.

If you remove the SET RS232 CONFIG. accessory though, you won't be able to set the transmission rate anymore. Fortunately there is a way to get around this inconvenience.

This rate can be set to operate at speeds from 50 to 19,200 baud. One way to set the transmission rate of this port is by writing a short machine language program.

Instead, we'll write a BASIC program which creates a short machine language program called SETBAUD.PRG to set the baud rate. You can execute SETBAUD.PRG as usual from the desktop.

As it stands, the program sets the transmission rate to 50 baud. To change the speed, change the value at the end of line 120 from 15 to one of the code numbers below:

```
10   rem *** Configure Baud Rate 5.1***
20   for i=1 to 72 step 2      : rem 36 Words
30   read a                 : rem Read
40   poke i+1999,a          : rem into memory
50   next i                 : rem hold
60   bsave "SETBAUD.PRG",2000,72:rem store
```

```
70    end
80    rem --- Program Data ---
100   data 24602,0,44,0,0,0,0,0,0,0,0,0,0,0
110   data 16188,-1,16188,-1,16188,-1
120   data 16188,-1,16188,-1,16188,15
130   data 16188,15,20046,-8196,0,14
140   data 16999,16188,76,20033
```

Code Number	Baud Rate
0	19,200
1	9,600
2	4,800
3	3,600
4	2,400
5	2,000
6	1,800
7	1,200
8	600
9	300
10	200
11	150
12	134
13	110
14	75
15	50

All other settings of the interface, such as the transmission protocol or the parity, remain in the condition set.

You can change the transmission protocol in addition, by replacing the second -1 in line 120 with one of the following values:

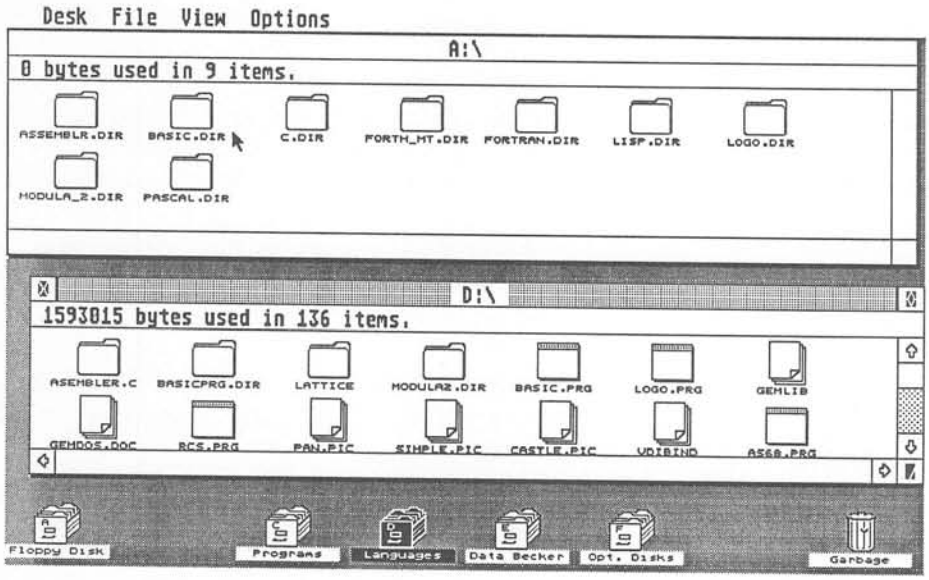
Value	Significance
- 1	Set value remains
0	No Handshake
1	XON/XOFF
2	RTS/CTS
3	Both, but does not make sense

Chapter 6

Programming Languages

Programming Languages

A single programming language must have an enormous scope to serve all the features of a computer of the caliber of the ST. For this reason the best was barely good enough for the Atari. There are a wide variety of languages available to the ST owner. Some of these languages should be examined closely to determine their advantages and disadvantages. Let's begin with the LOGO and BASIC interpreters, which are well suited for simple programs, and furthermore are easy to use.



6.1 DR.LOGO

The name of this programming language is often written as in this title, but this doesn't mean that it is a language for doctors. DR LOGO, as it should really be written, is an abbreviation for Digital Research LOGO, which we will call ST LOGO. This language, which also runs on other computers, is often considered a children's language. In reality, DR LOGO is a complete and capable language.

LOGO has some similarity to FORTH, which has a reputation of being an extensible programming language. The reason for this is that you can define new commands. This is also possible in LOGO. Let's look at an example.

However, its not easy to output data from a LOGO program to a printer. LOGO only has the commands `COPYON`, or `COPYOFF`, which control all the output to the printer. If you want to have the command `LPRINT`, which works like the one in BASIC, you can define it as follows:

```
TO LPRINT :LINE
COPYON PR :LINE COPYOFF
END
```

These statements, called *procedure*, are called by its name. The procedure `LPRINT[a]` is now available from the LOGO language. Text is output to the printer with `LPRINT[Text]`.

This example also points out a principle of LOGO programming. For simplicity in programming, we write procedures. These procedures are used by higher level procedures, and so forth, until the program has been finally reduced to a single command. This hierarchical structure has the advantage that you can program with a good overview, but requires that you follow a certain discipline for program structure. A program without a good structure will become nearly illegible.

Another principle of the LOGO philosophy is the list orientation of the language. Most of the data is managed as *lists* and list processing is heavily supported. Lists can consist of any number of numeric values or strings which can be sorted, mixed, checked or changed with a single command. Because of this ability, LOGO is suitable for text processing or spreadsheets.

LOGO has another specialty—graphics. It is also known as *turtle-graphics* since the cursor used for design in some LOGO versions has the shape of a turtle. The ST turtle is represented as a triangle which can be put into motion with commands such as LEFT, RIGHT, FORWARD or BACKWARD. If you tell it to, it will leave a trail as it moves, making it easy to produce designs. An example:

```
TO BROOCH
FD 100 RT 111  FD 40
TURN
BROOCH
END
TO TURN
FD 66  RT 66 FD 66
END
```

This example will paint a brooch on the display with the command BROOCH. Since the procedure BROOCH will call itself constantly, it will run until stopped, or until the power is turned off.

An exhausting description of the DR LOGO programming language would exceed the boundaries of this book. Those interested can consult a number of good books which introduce this language, such as *Atari ST LOGO User's Guide* from Abacus Software. Instead let's quickly look at how LOGO can be used to fool the operating system.

The PEEK and POKE functions exist here too, but they are called .EXAMINE and .DEPOSIT.

For example, .EXAMINE 1102 provides the address of the video display memory. This command works exactly like its BASIC counterpart PEEK.

.DEPOSIT 1262 starts the printout of the display. You can use this command just as you would the POKE command in BASIC.

6.2 ST BASIC

Among computer hobbyists, BASIC is probably the most popular language. This is because the language is easy to learn and is not very critical concerning programming style. A BASIC program can be written quickly and errors can be found fast using the trial-and-error principle, since you can correct errors occurring during execution immediately. This is not as easy with other languages such as C or machine language.

The ST BASIC interpreter stores the program lines as text. These text lines are interpreted line by line. Other BASIC interpreters work somewhat differently. With them a BASIC command is stored as an encoded number, called a *token*. This method simplifies the command interpretation and improves the speed of the program. But ST BASIC is relatively fast even without the preliminary translation to tokens.

A BASIC program can be loaded directly from diskette by a text editor and changed. Since a BASIC program is stored as text, it can be loaded and edited with most word processors. But since ST BASIC has a built in editor, using a word processor is usually unnecessary. The BASIC editor also performs a syntax check of each program line as it is entered.

BASIC is not a problem oriented language. While COBOL is designed for business and FORTRAN for scientific problems, BASIC is general and can be used for all applications. It does not offer special commands such as sorting or indexing, but can perform these functions with subroutines. Therefore, almost every problem can be solved with a BASIC program. A problem which is very difficult to do in BASIC is the handling of real time processes. For time critical assignments, machine language is much better suited. You can, however, combine these two languages, which we'll look at Section 6.4.1.

This book is also written for the BASIC programmer who wants to fully experiment with the fabulous possibilities of the ST. BASIC commands such as `PEEK, POKE or CALL` permit accessing of the operating system facilities. For this reason the book was not called "`EXAMINES and DEPOSITS`" as `PEEK` and `POKE` are called in LOGO, since only BASIC permits almost complete access to the ST.

6.3 The C Language

The C programming language is a high level language whose strange name is derived from the fact that its developers named the earlier versions of the language by alphabetic characters. First came A, then B, and finally C. It is now a widely used language on many computer systems.

Although it's considered a high level language, there are some elements of the language which make it very close to machine language. This is especially apparent since input and output functions are not part of the standard language definition. This is because these functions are very system specific, i.e. dependent upon the computer being used. To perform input and output, each C compiler contains a library of input and output functions which are then linked to the user's program.

One advantage of the C language is that it is a transportable language. C programs written for one computer are often easily adapted for use on a different computer.

The ST is a good example of such a computer. GEM for the ST is written in the C language. The same version of GEM is available to run on IBM/PC compatible computers, thus demonstrating C's transportability.

Another example of this transportability is the powerful operating system UNIX, which is mainly used in large computer systems. It too is written in C and is available on many different computers. You can see that the C language is quite powerful!

C is derived from the family of the ALGOL language. From this language family both FORTRAN and PASCAL originated. These languages are characterized by their high popularity and structured approach to programming. You can define procedures whose use is similar to a new command. The procedure `PRINTF` is one of these. It belongs to the standard input/output routines which are made available by the above mentioned library. This procedure outputs, similar to the `PRINT` command in BASIC, a text or numeric value. The text or values to be output are passed to the function with formatting directions for the output. Here's a sample call to this function:

```
PRINTF("5.2f %6.1f\n",A,B)
```

As complicated as this expression appears, it is simple to understand. The arguments which begin with % determine the format of the output. %5.2f means that the value of the variable A should be represented as a floating point number with at least 5 characters width and 2 positions after the decimal point. The f stands for floating point.

Let's return to the normal language definition of C. It contains only simple control structures such as decisions, loops and subprograms. Variables must be defined at the beginning of a program to specify the data type. A simple and unstructured programming style like BASIC's is not permissible here. This programming style also makes error detection easier.

The structure of a C program is as follows:

First the program variables are defined. Then individual functions are written. To follow the execution of a C program, first find the routine with the name main(). This is the real program which contains a series of function calls. Since the names of the functions usually indicate their purpose, the general structure of the program can be easily determined.

Small programs often consist only of the main routine. Let's look at a small C program.

```
/* Change of Radius to Circumference */
main()
{       int from, to, step;
        float radius, circumference;
        from = 1; /* Beginning Radius */
        to = 10;  /* End Radius */
        step = 1  /* Increment */
        radius = from;
        while (radius <= to)
        {
      Circumference = 2*radius * 3.141592;
      printf("%2.0Of %7.3f\n",radius,circumference);
      radius = radius + step;
        }
}
```

The first line represents a comment. A comment is enclosed within the characters /* and */ and is ignored by the compiler. After that our

program starts with the MAIN() routine. The empty parentheses indicate that this function does not require a parameter.

Next follow the definitions of the variables. The following data types are permissible:

int	Integers
float	Floating point
char	A single character
short	A small, whole number value
long	A large, whole number value
double	A floating point value with double precision

The program assigns initial values to the variables. After that we start the loop to calculate the 10 circumference values. This loop is defined by WHILE(*condition*) and continues as long as the condition is met. A WEND, as required in BASIC for the completion of a WHILE loop, does not exist in C. The program portion which is enclosed by the braces following the statement is repeated by this command.

The program portion calculates the circumference of a circle from the radius and outputs both values in formatted form. Then the radius is increased by one and the loop repeated if it has not exceeded the end value of 10. If the radius is larger than 10 the program is terminated.

As you can see, a C program is not very hard to follow if you have already learned another structured programming language. In C, all statements are terminated with a semicolon, as in PASCAL. The keywords IF, ELSE, WHILE, or FOR are all familiar. The somewhat cumbersome programming becomes fairly readable through structuring and the amount of work required is rewarded with the high computation speed of the completed program.

For the ST there is a another reason to use C. Accessing GEM is made easier since the parameters are well defined for the C programmer. The ST development package from Atari contains a giant library of GEM functions and systems parameters which make all of GEMDOS accessible to the C programmer.

6.4 68000 Machine Language

In the world of programming languages, *machine language* occupies a special position. This language is at the same time the most primitive and most powerful for programming a computer. This seeming contradiction is easy to explain.

Every computer contains a central processing unit or CPU. This "brain" of the computer is the building block that really does all the work which is assigned to the computer. If you program in the higher level languages such as BASIC or LOGO, the input is translated into the language the processor can understand. This language is machine language. The advantage it has in comparison with other languages is that it can control all the functions of the computer. This is why we are often forced to use machine language.

We discovered one of these limitations in the section where we configured the serial interface. Even the general command `POKE` was not sufficient, and we had to write a machine language program. This is only the tip of the iceberg.

Another advantage of machine language programming is the speed that we can achieve. You only have to look at the speed with which the screen is erased. A program in BASIC would appear as follows:

```
5     rem*** basic.bas 6.4***
10    defdbl a,b : b = 1102
20    a = peek(b)
30    for i = a to a + 32768
40    poke i,0
50    next i
```

Those who can estimate the speed of a BASIC program can see immediately that even a fast BASIC, such as the ST's, can take a few seconds to run this program. You can watch as the display is erased line by line. Erasing the screen with a machine language routine on the other hand, will proceed faster than you can watch.

You would think that the operating system would support such procedures to relieve you of such time problems. But what about

applications such as word processing where a sort routine should not require a forced coffee break?

You will find that you cannot ignore machine language if you want to make fullest use the system. Especially since the 68000 processor used in the ST also offers a powerful operating system such as GEMDOS.

Let's first examine the processor and its language. There are no variables or such easy commands as `PRINT 16^3`. The programmer has a few instructions at his disposal, but they concern themselves only with bits, bytes, words, long words and registers. If you want to store a value, it has to be stored in the working storage location and you have to remember the exact location.

Don't be scared off, it's not as complicated as it sounds. Let's consider an example. Assume that two data words are located in locations $1000 and $1002 and they must be added. The result will be stored in location $1004. A machine language program would appear as follows:

```
MOVE.W $1000,D0
ADD.W D0,$1002
MOVE.W D0,1004
```

That's it! The `MOVE` instruction moves data in the computer. The extension `.W` indicates that it concerns a word, that is 16 bits. This could have been `.B` for a byte or `.L` for a longword. The first instruction of our program takes the word from storage location $1000 and puts it into D0. This D0 is a data register, a 32 bit storage location in the computer. The 68000 has 8 of these registers, designated D0 to D7. Data which are being worked on can be stored in these registers. Registers have additional advantages in comparison with working storage, which won't discuss at this time.

The data register D0 now contains the value of the storage location $1000. This value is added to the content of $1002 with the `ADD.W` instruction. Here too we append `.W` for the actual data length, a word. The result of the addition is now in D0, from where it will have to be transferred to the desired storage location. For this we'll again use the `MOVE.W` instruction. The `MOVE` instruction always transfers the value of the first referenced storage location (or register) to the second. A common method of writing this is as follows:

```
MOVE.datatype from, to
```

Datatype means the letters .B, .W, or .L. From and to indicate from where and to where the data are to be transferred. For this type of addressing, the 68000 offers a large selection. The following table contains the addressing modes:

Addressing Mode	Example
Register direct:	
- Data register direct	CLR D0
- Address register direct	MOVEA A1,A6
- Status register direct	MOVE D0,SR
- Program counter direct	MOVE D0, CCR
Data immediate:	
- immediate long	MOVE#$20000, D0
- immediate short	MOVE #$20,D0
- immediate quick	MOVEQ #9,D0
Absolute:	
- absolute short	NOT $2000
- absolute long	NOT $18000
Address Register indirect:	
- simple	CLR (A0)
- with displacement	CLR 80(A0)
- with displacement and Index	MOVE 8(A0,D0),$400
- with Predecrement	CLR -(A7)
- with Postincrement	CLR (A7)+
Program Counter relative:	
- with displacement	MOVE 4(PC),A0
- with displacement and Index	MOVE 8(PC,A0),D0
- Branch command	BRA Label

As you can see from the table, addressing modes can be differentiated in 5 categories. The first three are easy to understand:

Register direct: here a register is the direct source or destination whose content is moved or changed. The possible registers are the data registers D0 to D7, the address registers A0 - A7, the Status Register (SR) and the Condition Code Register (CCR). Use caution with A7, since this register is also the Stack Pointer (SP).

Data immediate: the data is part of the instruction. This addressing mode is only suitable for a data source. This addressing mode is similar to the `POKE` command, since `MOVE #10,1000` is just like `POKE 1000,10`. It writes a fixed value (10) into a memory location.

Absolute: the directly provided address is addressed and its content used. With `PEEK` and `POKE` only absolute addresses can be addressed. There are other methods of selecting a certain memory location. These addressing methods are sometimes rather complex, but very effective.

Now let us consider two other methods of addressing.

Indirect Addressing: Here the register used is not addressed directly, but its content is interpreted as an address. With this method simple pointers can be processed by loading the value of the pointer into a register and addressing indirectly. An example:

```
MOVE.L 1102,A0          * Screen address in A0

MOVE.W #1,(A0)          * and display "." on screen
```

This small machine language program can be done in BASIC only with a lot of overhead. It would appear as follows:

```
5     rem*** basic2  6.4***
10    defdbl a,b       : rem Long Words defined
20    b = 1102 : rem b as Address for Pointer
30    a = peek(b): rem Determine display addr
40    poke a,1         : rem and display
```

This short example shows the advantage of indirect addressing. You can specify a displacement or offset which is added to the address contained in the register to produce a different effective address.

In the statement:

```
MOVE.W    #1, N(A0)
```

the offset n is added to the contents of address register A0. The immediate value 1 is then stored at this effective address.

Since the address of the start of the video display memory is already contained in address register A0, we can display a point on the second line using an offset of 80 as follows:

```
MOVE.W    #1, 80(A0)
```

The offset is limited to ±32767.

There is also a variation of indirect addressing using either postincrementing of predecrementing. Using these modes, the specified register is either incremented after (post) or decremented before (pre) the operation. Using these variations, it is easy to work with stacks.

If a word is to be stored on the stack you could use the following instruction:

```
MOVE.W    #0000,-(SP)
```

The stack pointer is decremented by two after the value 0000H is pushed onto the stack. Similarly you can retrieve the word form the stack with this instruction:

```
MOVE.W    (SP)+,D0
```

The data is moved from the top of the stack into data register D0 and the stack pointer is incremented by two. The stack is now ready to accept more data.

Relative addressing: The last addressing mode depends on the contents of the program counter. The program counter contains the address of the machine language instruction which is currently being executed.

One variety of relative addressing is the branch instruction. A branch instruction alters the path of program execution either conditionally or

unconditionally. The conditional branches are usually based on the outcome of a previous instruction. For example:

```
CMP.W #10,D0

BLT   SMALLER
```

Here a branch to the program section 'SMALLER' is made if the content of the register D0 is less than 10. BLT means specifically 'Branch if Less than' which is self explanatory. The following table contains all branch instruction variations:

Mnemonic	Condition	Branch if...
BEQ	equal	Z = 1
BNE	not equal	Z = 0
BPL	plus	N = 0
BMI	minus	N = 1
BGT	greater than	Z+(N&V) = 0
BLT	less than	N&V = 1
BGE	greater or equal	N&V = 0
BLE	less or equal	Z+(N&V) = 1
BHI	higher	C + Z = 0
BLS	lower or same	C + Z = 1
BCS	Carry set	C = 1
BCC	Carry Clear	C = 0
BVS	Overflow	V = 1
BVC	no overflow	V = 0
BRA	branch always	
BSR	branch to subroutine	

The + sign indicates logical OR, & stands for logical AND (See section 3). The characters in the right column of the table indicate the flag bits of the condition code register (CCR) that may be set by each operation. Each of these bits has its own significance and is influenced by the different instructions.

Some of these instructions alter the flow execution within the program. The JSR instruction (Jump to SubRoutine) is similar to the BASIC GOSUB command. When a JSR instruction is performed the path of execution is temporarily altered until an RTS instruction (ReTurn from

Subroutine) is encountered. Likewise, the JMP instruction is analogous to the BASIC GOTO command.

The TRAP instruction is another instruction which alters the processing sequence. TRAP#1 calls the GEMDOS, TRAP#13 calls the BIOS and TRAP#14 calls the XBIOS. These instructions are usually used to access the built-in operating system routines.

Here is a list of program control instructions for the 68000.

Mnemonic	Significance
Bcc	Branch if condition code true
BRA	Branch always
BSR	Branch to a subroutine
CHK	Check register against limits
DBcc	Test condition, decrement and branch
JMP	Jump to address
JSR	Jump to subroutine
NOP	No operation
RESET	Reset peripherals
RTE	Return from an exception
RTR	Return and restore register
RTS	Return from a subroutine
Scc	Set a byte according to condition code
STOP	Stop with condition code loaded
TRAP	Software trap always
TRAPV	Trap on overflow

In the short sample program in which two numbers were added, an arithmetic operation was performed: addition. Besides addition, the 68000 can also perform subtraction, multiplication and division. These operations are often missing from the instruction sets of the earlier processors.

Here is an overview of the arithmetic operations:

Mnemonic	Significance
ADD	Binary addition
ADDA	Add binary to address register
ADDI	Add immediate
ADDQ	Add immediate quick
ADDX	Add binary with extended
CLR	Clear
CMP	Compare
CMPA	Compare address register
CMPI	Compare immediate
CMPM	Compare in memory
DIVS	Divide with sign
DIVU	Divide without sign
EXT	Extend sign
MULS	Multiply with sign
MULU	Multiply without sign
NEG	Negate
NEGX	Negate with extend
SUB	Subtract binary
SUBA	Subtract binary address register
SUBI	Subtract immediate
SUBQ	Subtract immediate quick
SUBX	Subtract binary with extend
TST	Test byte

In addition, the 68000 can also process BCD numbers. For this it has the following instructions:

Mnemonic	Significance
ABCD	Add BCD numbers with extend
NBCD	Negate BCD number
SBCD	Subtract BCD numbers with extend

Next the logical operations which we're familiar with from BASIC:

Mnemonic	Significance
AND	Logical AND
ANDI	Logical AND with immediate value
EOR	Exclusive OR
EORI	Exclusive OR with immediate value
NOT	Logical NOT
OR	Logical OR
ORI	Logical OR with immediate value
TAS	Test byte and set always bit7

Single bits can be manipulated directly using the following instructions:

Mnemonic	Significance
BCHG	Test bit and change
BCLR	Test bit and clear
BSET	Test bit and set
BTST	Test bit

The processor can also shift and rotate the bits in an operand.

Mnemonic	Significance
ASL	Arithmetic shift left (*2)
ASR	Arithmetic shift right (/2)
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right
ROXL	Rotation left with extended bit
ROXR	Rotation right with extended bit

And now we come to the instructions which move data in the computer:

Mnemonic	Significance
EXG	Exchange registers
LEA	Load an effective addr to addr register
LINK	Link local base pointer
MOVE	Move source data to destination
MOVE from SR	Transfer the content of the SR
MOVE to CCR	Move flags to CCR
MOVE USP	Move user stack pointer
MOVEA	Move to address register
MOVEM	Move multiple register
MOVEP	Move to or from peripheral register
MOVEQ	Move immediate quick
PEA	Push effective on stack
SWAP	Swap register halves
UNLK	Unlink local area

These then, are the instructions of the 68000. In combination with the various addressing modes, these instructions can be used to make programs as efficient as possible.

Now we want to examine the program that we used to convert the display scan rate form 60 Hertz to 50 Hertz. The machine code version of the program is as follows:

```
CLR.L -(SP)
MOVE.W #$20,-(SP)
TRAP  #1              * Set Supervisor State
ADDQ.L #6,A7          * Adjust Stack
MOVE.B #2,$FF820A     * Set PAL Frequency
MOVE.L D0,-(A7)       * alter SSP
MOVE.W #$20,-(SP)
TRAP  #1              * Set User-State
ADDQ.L #6,SP          * Adjust Stack
CLR.W  -(SP)
TRAP  #1              * Back to Desktop
```

The first TRAP changes the system to the supervisor state. We need to do this since access to the I/O area is privileged and would generate a bus-error in the user mode (and two cherry bombs...). After adjusting the stack pointer, we can change to the 50 Hertz frequency. TRAP #1 calls

GEMDOS again and passes the original value of the supervisor stack pointer which puts the computer system back into user state. The state is adjusted again and control is returned to the desktop.

The BASIC equivalent of the machine language programs which follow contains some additional values which precede the application program. These values are not part of the program, but are required by GEMDOS to recognize the length and kind of program. If we call a machine language program from BASIC with CALL, these values are not required. Using TRAP #1 with the function number 0 is sloppy programming, since GEMDOS does not return to the BASIC program, but to the desktop. Such a program is generally ended with the RTS instruction, which returns control to BASIC.

6.4.1 Combining Machine Language and BASIC

The advantages of machine language are so great that it would be very interesting to combine it with the easier to write BASIC programming language. There are a few commands in BASIC for this and we already know about PEEK and POKE. To be able to access a machine language program command from BASIC, you can use the CALL command. This command allows you to pass parameters to the machine language program.

Let's first consider the CALL command. To do this we'll examine a program to set the baud rate of the serial interface directly from a BASIC program. In the earlier section on setting the baud rate, we wrote a similar program which you could access only from the desktop. Here's another version:

```
10   rem *** Configure Baud Rate 6.4.1***
12   a$=space$(40)   : rem Reserve Memory
14   b=varptr(a$): rem Determine Address
20   for i=0 to 36 step 2: rem in a Loop
30   read a          : rem read data in
40   poke b+i,a      : rem and store
50   next i
60   input "Baud rate-code (0 - 15)";x: rem
     enter code
```

```
70   poke b+22,x              : rem store
80   call b                   : rem and call
90   end                      : rem that's it!
100  rem -- Machine Program --
110  data 16188,-1,16188,-1,16188,-1
120  data 16188,-1,16188,-1,16188,15
130  data 16188,15,20046,-8196,0,14
140  data 20085
```

As you can see, the machine language program is considerably shorter than the preceding program. This is because the program paramater table must precede every application program so that the GEMDOS can determine the beginning and ending address of the program. Furthermore, the program must end with a special GEMDOS call where control is returned again to the desktop or the calling program.

When using the CALL command, no preset parameters are required. A simple RTS instruction in the machine language program (RTS corresponds to the BASIC command RETURN) is sufficient to return control to the BASIC program that issued the CALL.

The machine language program is read from DATA statements and POKEd into memory. One characteristic of the sample program is that this memory area is movable. For this reason the text variable A$, whose location is determined by the BASIC interpreter becomes the storage area for the machine language program. This is a great advantage. In this manner several machine language programs can be integrated into a BASIC program without having to worry about their memory locations. Using a fixed memory area can create problems if GEMDOS or GEM also use this area and destroy the machine language program. CALLing such a program in a fixed location usually leads to a crash.

The machine language program is POKEd into the area reserved for the variable A$ after the FOR-NEXT loop. Next the computer asks you to enter a code for the baud rate. Any code may be entered, but the 16 possible baud rates are normally sufficient to cover all needs for speed in data transmission. You simply enter the code number for the desired baud rate, ranging from 0 (19,200 baud) to 15 (50 baud). The table is found in section 5.1 of this book.

The value entered is now passed to the machine language program by POKEing into the corresponding location. Next the machine language

program is accessed with CALL B, where B is the address of the variable and therefore the programs. The program calls the extended BIOS with a TRAP instruction and passes the parameters to set the RS-232 port. This machine language program looks as follows:

```
          * Setting Baud Rates *

MOVE.W #-1,-(SP)          * Synchron - Character
MOVE.W #-1,-(SP)          * Transmit Status Register
MOVE.W #-1,-(SP)          * Receiver Status Register
MOVE.W #-1,-(SP)          * USART Control Register
MOVE.W #-1,-(SP)          * XON/XOFF and RTS/CTS
MOVE.W #15,-(SP)          * Baud rate (BSPL, 50 Baud)
MOVE.W #15,-(SP)          * Command

TRAP  #14                 * XBIOS - Call
ADD.L #14,SP              * Correct Stack
RTS                       * Return to BASIC
```

The 15 in line 6 which determines the baud rate, is the number which is changed with the POKE command. By adding another POKE command to the BASIC program you can change the transmission protocol (XON/XOFF and RTS/CTS). Do this with POKE B+18,n to set a new protocol. The -1 parameter means that the original setting is to be retained.

This example illustrates the CALL command as a call for a machine language program. The repeated use of CALL B assumes that the A$ variable still contains the program and the B its address. Now we can change the addresses of these variables with the BASIC program changes. It would be wise to determine the condition of the variable B before every call.

You can also pass parameters to a machine language program by enclosing them within parentheses. For example, CALL A(1,2,3). When you do this, BASIC puts these parameters onto the user stack with your machine language program.

The following example passes three parameters to a machine language routine. The machine language routine simply moves the parameters to an unused area of memory starting at $2000 (8192 decimal). This machine language routine simply illustrates the passing of values from BASIC.

BASIC portion:

```
10   rem *** Calltest 6.4.1***
20   a$=space$(100)       : rem Make space
30   a=varptr(a$)    : rem Determine address
40   bload "calltest.prg",a:rem load program
90   rem End of first part
100  call a (1,2,3)       : rem test call
130  n=peek(8192)    : rem Determine number
     (2000 hex)
140  print n;"Arguments :"
150  defdbl j             : rem Set Longword
155  d=1                  : rem Initiate counter
160  for j=8208 to 8204+n*4 step 4
170  print d;": ";peek(j): rem Output
     Parameter
180  d=d+1                : rem Increment Counter
190  next j               : rem continue
```

Machine language portion:

```
*  calltest.prg    6.4.1
     MOVE.W    4(A7),$2000      * Save # of param.
     MOVE.L    14(A7),$2010     * First Parameter
     MOVE.L    18(A7),$2014     * Second Parameter
     MOVE.L    22(A7),$2018     * Third Parameter
     MOVE.L    26(A7),$201C     * Fourth Parameter
     MOVE.L    30(A7),$2020     * Fifth Parameter
     RTS
```

You'll have to assemble the short machine language routine before running it. If you don't have an assembler we've included a short loader to create "calltest.prg."

The BASIC portion first BLOADs the machine language portion and then immediately CALLs it and passes three parameters. To verify that the machine language program works correctly, lines 130-190 PEEK the area of memory to which the parameters were moved.

Here is a BASIC loader to create the `calltest.prg` on diskette:

```
1000 open"R",1,"calltest.prg",16
1010 field#1,16 as bin$
1020 a$=""; for i = 1 to 16:read d$: if d$="*" then 1050
1030 a = val("&H"+d$): s=s+a:a$=a$+chr$(a):next
1040 lset bin$=a$:rec=rec+1:put 1,rec:goto 1020
1050 data 60,1A,00,00,00,60,00,00,00,00,00,00,04,00,00
1060 data 00,00,00,00,00,00,00,00,00,00,00,00,33,C0,00,00
1070 data 20,00,23,EF,00,0E,00,00,20,10,23,EF,00,12,00,00
1080 data 20,14,23,EF,00,16,00,00,20,18,23,EF,00,1A,00,00
1090 data 20,1C,23,EF,00,1E,00,00,20,20,4E,75,23,DF,00,00
1100 data 00,60,4E,4E,2F,39,00,00,00,60,4E,75,23,DF,00,00
1110 data 00,60,4E,4D,2F,39,00,00,00,60,4E,75,23,DF,00,00
1120 data 00,60,4E,41,2F,39,00,00,00,60,4E,75,00,00,00,32
1130 data 08,08,08,08,08,00,00,00,00,00,00,00,00,00,00,00
1140 data *
1150 close 1: if s <> 5243 then print "ERROR IN DATA!":end
1160 print"OK"
```

Chapter 7

BASIC Programming

BASIC Programming

BASIC is familiar to most of us because it is easy to learn and can be used for so many applications. With most of the BASIC commands you can write programs and make changes to them without fear of crashing the computer.

But when it comes to the PEEK and POKE commands, you may not be as secure. These commands involve access to memory at the machine language level. An erroneous POKE may crash the computer. Before experimenting with PEEK and POKE make sure that you have SAVEd any programs in memory in case the program crashes.

Load the BASIC interpreter and let's begin to program.

The programs presented here contain many PEEK and POKE commands, just as you might suspect from this book's title. If you've worked with BASIC before, you'll have no trouble using these two commands. PEEK examines a memory location and POKE modifies a memory location.

There is one point that you should be aware of when using ST BASIC. The 68000 processor in the ST is capable of accessing memory in increments of bytes, words or longwords. The PEEK and POKE commands of ST BASIC are also capable of accessing bytes, words or longwords at a time. Therefore you must specify the size increment to be used with PEEK or POKE.

Let's assume that you want to set the byte at memory location 8000 to zero. By default, ST BASIC is set to access a word (two bytes) of memory at a time. If we POKE 8000,0 not only is the memory at 8000 set to zero, but also the memory at 8001.

To solve this problem, you can use the following technique:

```
DEF SEG = 1
X = PEEK(A)
```

The DEF SEG statement tells BASIC that the offset for any access to memory is one byte. An offset is equivalent to adding 1 to any address. Therefore this sequence of commands DEF SEG = n:PEEK(A) reads the address A+n. Also, all accesses to the address A are in byte

increments. Keep in mind that all subsequent accesses to memory use the offset n, therefore you must reset the offset to zero when you're finished by using DEF SEG = 0.

Another problem appears when we want to modify a pointer in storage. Recall that a pointer is always a longword of 4 bytes. We could use two POKE commands to change such a pointer completely. This method has two problems. The number which is to be stored in the pointer must be divided into the higher and the lower word which requires additional work. Furthermore, the pointer's value may change during access of the two POKE commands. This can have an unfortunate effect if the pointer is used in an interrupt routine which may occur at a point when one POKE has been completed and not the other. At that point the pointer is in an undefined condition and may lead to a system crash when used by an interrupt routine.

To overcome this problem you can use the DEFDBL statement. For example, the pair of statements DEFDBL A: POKE A,X tells BASIC to access the memory specified by variable A as a longword, or 4-bytes.

To explicitly specify that memory is to be accessed as a word, use the statement DEFSNG B. Now all accesses to the memory specified by variable B are as words, or 2-bytes.

The technique of controlling the access width of PEEK and POKE commands makes programming very flexible. But you must remember to respecify the width explicitly. We frequently made the error of not respecifying the defaults and our programs crashed on subsequent tests. The RUN command does not reset the access widths.

If you want to experiment with PEEK and POKE in the following programs, pay attention to the exact definition of the access width. A completely defined program cannot function if it processes bytes instead of words. To be absolutely sure, write the command DEF SEG = 0 into the first line of a program which uses PEEK and POKE.

7.1 Graphics

The ST is well equipped for graphic processing with its high resolution display and the GEM operating system. GEM supports nearly everything a programmer may want to use for graphics. You can draw lines, paint circles and ellipses, squares with square or round corners, and fill in various forms with shading using simple procedures. Let's start now by drawing a shaded circle.

Perhaps you have drawn circles on other computers using the sine and cosine functions. With ST BASIC we don't have to do this. The command `PCIRCLE X,Y,R` does this quickly. The command has a limitation. It can only draw a circle in the `OUTPUT` window of the BASIC display.

If we use GEM for the graphics functions, we don't run up against this limitation. With GEM we can draw anything anywhere on the screen. We also have additional capabilities beyond BASIC.

7.1.1 Circles, Ellipses and Squares

To draw a circle on the screen we have to use a `VDISYS` call which we used in an earlier chapter. The circle is drawn and filled in with the following program:

```
10   rem *** Draw shaded Circle 7.1.1***
20   color 1,1,1,1,1     : rem shading color
30   poke contrl,11      : rem Draw circle
40   poke contrl+2,3: rem Parameter number
50   poke contrl+6,0
60   poke contrl+10,4 rem Functions - ID
70   poke ptsin,100:rem X-center coordinates
80   poke ptsin+2,100 : rem Y-coordinates
90   poke ptsin+4,0       : rem Dummy
100  poke ptsin+6,0       : rem Dummy
110  poke ptsin+8,50      : rem Radius
120  poke ptsin+10,0      : rem Dummy
130  vdisys 0  : rem and normalize execution
```

The function 11 which is passed in line 30 to the `CONTRL(0)` field can do more than just draw a circle. With this one function 10 different graphics can be drawn. The information concerning what pictures are to be drawn, is contained in the function ID which is passed in `CONTRL(5)`. The assignment of ID numbers to the drawings is as follows:

ID	Drawing
1	Shaded Rectangle
2	Section of Circle
3	Shaded Circle Section
4	Shaded Circle
5	Ellipse
6	Section of Ellipse
7	Shaded Section of Ellipse
8	Square with rounded corners
9	Shaded Square with rounded corners
10	Justified text

The first of these functions, the shaded rectangle, is suitable for the creating bar graphs. Bars of any width and shading can be drawn. Such a rectangle can be created with the following program:

```
10    rem *** Shaded Square 7.1.1***
20    color 1,2,2,9,2    : rem shading attribute
30    poke contrl,11     : rem Shade square
40    poke contrl+2,2    : rem Parameter number
50    poke contrl+6,0
60    poke contrl+10,1   : rem Functions - ID
70    poke ptsin,50      : rem X-first Coordinate
80    poke ptsin+2,50    : rem Y-first Coordinate
90    poke ptsin+4,100   : rem X-second Coordinate
100   poke ptsin+6,200   : rem Z-second Coordinate
110   vdisys 0      : rem and normalize execution
```

The first coordinates are the upper left corner of the square and the second is the lower right corner. The significance of the shading attributes are explained later.

The next function, a section of a circle, corresponds to the BASIC command `CIRCLE X,Y,R,A,E` and draws an unshaded section of a

circle on the display. To do this the beginning and ending angle of the circle are specified. These angles are given in 1/10 degrees where the zero degrees lies to the right of the midpoint. A quarter circle in the first quadrant, right of the midpoint, lies therefore between 0 and 900 (90 degrees). The same is true of the CIRCLE command. An example:

```
10    rem *** Circle Section 7.1.1***
20    color 1,0,2,1,1    : rem Line attributes
30    poke contrl,11     : rem Draw Circle Section
40    poke contrl+2,4    : rem Parameter number
50    poke contrl+6,2
60    poke contrl+10,2   : rem Function - ID
70    poke ptsin,150     : rem X-center point
80    poke ptsin+2,50    : rem Y-center point
90    poke ptsin+4,0          : rem Dummy
100   poke ptsin+6,0          : rem Dummy
110   poke ptsin+8,0
120   poke ptsin+10,0
130   poke ptsin+12,30   : rem Radius
140   poke intin,0       : rem Beginning Angle
150   poke intin+2,900   : rem End Angle
160   vdisys 0           : rem and normalize execution
```

The same program can also draw a shaded circle section. To do this you must insert function ID 3 and the desired shading attributes in line 20. To draw a complete circle don't use 0 to 360 degrees with function 3, just use function 4. It has already been presented in our first example.

Now to the ellipses. For this, functions 5,6 and 7 may be used. Let's take function 6 as an example. This draws an ellipse section whose ends are connected with the midpoint.

```
10    rem *** Ellipse - Section 7.1.1***
20    color 1,0,1,1,1  : rem Line attribute
30    poke contrl,11:rem Draw Ellipse section
40    poke contrl+2,2  : rem Parameter number
50    poke contrl+6,2
60    poke contrl+10,6 : rem Functions - ID
70    poke ptsin,200   : rem X-Mid Coordinate
80    poke ptsin+2,100 : rem Y-Mid Coordinate
90    poke ptsin+4,100    : rem X-Radius
100   poke ptsin+6,50     : rem Y Radius
```

```
110  poke intin,3200      : rem Start Angle
120  poke intin+2,1200    : rem Final Angle
130  vdisys 0  : rem and normalize execution
```

For function 7 only the function ID and shading attributes change. These can be set with the COLOR command. Function 5 on the other hand requires no starting or ending angle since it draws a complete ellipse. Lines 110 and 120 are scratched and not replaced. In Line 50 a 0 is inserted and the function ID is replaced with the 5.

Next comes a form which is difficult to do from a BASIC program—a rectangle with rounded corners. For this we have ID numbers 8 and 9 whose use is nearly identical. The difference between the two is that the number 9 is shaded and number 8 is not. Only the ID must be changed and the shading/line attributes of the COLOR command.

```
10    rem *** Rounded Rectangle 7.1.1***
15    clearw 2: fullw 2
20    color 1,2,2,3,4    : rem Shading attributes
30    poke contrl,11     : rem Draw Rectangle
40    poke contrl+2,4    : rem Parameter number
50    poke contrl+6,0
60    poke contrl+10,9   : rem Functions - ID
70    poke ptsin,50      : rem X-Start Coordinate
80    poke ptsin+2,50    : rem Y-Start Coordinate
90    poke ptsin+4,190 : rem X-Target Coordinate
100   poke ptsin+6,190   : rem Y-Target Coordinate
110   vdisys 0      : rem and normalize execution
```

We have only looked at the only the basic graphic functions which GEM easily accomplishes. But good control of graphics also includes text. The VDI is well suited for this since it controls text formatting. This is our next topic.

7.1.2 Text on the graphics screen

The last of the 10 calls of the VDI lets you put text formatted in many ways on the screen—at any location. This function takes several parameters. For one, there is the display position at which the text begins. These coordinates point to the upper left hand corner of the first character in the text. Furthermore, the desired total length of the text must be specified. If this is longer than the text, it will be adjusted by adding spaces.

Finally, the text itself is passed. It must be passed one character at a time to the INTIN array which makes the technique a little more difficult than the graphics programs above. Here is a sample:

```
10    rem *** Graphic Text - VDISYS- Demo 7.1.2***
15    clearw 2:fullw 2: rem clear the screen
20    text$ = "Sample text"
30    poke contrl,11           : rem Command code
40    poke contrl+2,3    : rem Parameter number
50    poke contrl+6,Len(text$)+2
60    poke contrl+10,10 : rem Functions - ID
70    poke intin,0       : rem Word stretching (0=out)
80    poke intin+2,0     : rem Character stretching
90    poke ptsin  ,50    : rem X Coordinate
100   poke ptsin+2,100   : rem Y Coordinate
110   poke ptsin+4,50    : rem X-Text length
120   for i=1 to len(text$) : rem ASCII characters
130   poke intin+2+j*2,asc(mid$(text$,i,1)) : rem set
140   next i
150   vdisys 0           : rem and normalize execution
```

Since we're already on the subject of text, we should examine the capabilities offered by GEM. In Section 4.2.4 we described how you could select a typical font of the ST. Sometimes you may want to select characters or numbers in various sizes and directions. This is where GEM helps. Let's examine a program that places giant characters on the screen:

```
10    rem *** Change Character Size 7.1.2***
20    poke contrl,12       : rem Command code
30    poke contrl+2,1: rem Parameter number
40    poke contrl+6,0
```

```
50   poke ptsin,0          : rem Dummy
60   poke ptsin+2,30: rem Character height
70   vdisys 0              : rem and execution
80   text$= "Fantastic text !"
90   poke contrl,8         : rem Text
100  poke contrl+2,2
110  poke contrl+6,len(text$)
120  poke ptsin,100         : rem X Coordinate
130  poke contrl+2,100     : rem Y Coordinate
140  for i=1 to len(text$)
150  poke intin+2*i-2,asc(mid$(text$,i,1))
160  next i
170  vdisys 0              : rem write text
180  poke contrl,12 : poke contrl+2,1
190  poke contrl+6,0 : poke ptsin+2,6 :
     rem ptsin+2,13 for mono
200  vdisys 0              : rem and normalize
```

Here are two new VDI commands, 8 and 12. VDI command 12 sets the height of the characters. This height is specified in `PTSIN(1)` and is the only parameter of the function. The 0 in the `PTSIN(0)` is not used but we set it for safety's sake.

The next section calls function 8 which writes text at the specified X/Y coordinates on the screen. Here a simple `PRINT` command is possible which writes only to the BASIC output window.

Finally, the standard size is restored. This must be done since the display editor does not know what to do with the enormous amount of text so that subsequent data to the `OUTPUT` window would otherwise produce only garbage.

Some interesting effects can be produced using this function. But what can we do if we need to label a vertical line? The VDI can also help do this. With function 13, the base line on which the text is written can be rotated in 90 degree steps. Zero (0) is the default setting. The angle is specified in 1/10th of a degree units. An angle of 900 permits rotating the text vertically to the left. 1800 turns it upside down and 2700 prints it vertically to the right. Unfortunately the monitor of the ST does not support any angle in between. These are reserved for other devices such as a plotter.

It can be done as follows:

```
10    rem *** Change Character Direction 7.1.2***
15    clearw 2: fullw 2
20    poke contrl,13          : rem Command code
30    poke contrl+2,0       : rem Parameter number
40    poke contrl+6,1
50    poke intin,900          : rem 90 degrees
60    poke ptsin2,30
70    vdisys 0                 : rem and execute
80    text$= "Vertical text !"
90    poke contrl,8           : rem Text
100   poke contrl+2,2
110   poke contrl+6,len(text$)
120   poke ptsin,100
130   poke ptsin+2,160
140   for i=1 to len(text$) : rem Transmit
      text
150   poke intin+2*i-2,asc(mid$(text$,i,1))
160   next i
170   vdisys 0                  : rem write text
180   poke contrl,13          : poke contrl+2,0
190   poke contrl+6,1         : poke intin,0
200   vdisys 0                  : rem and normalize
```

Here the text output function 8 was used. The normal PRINT command is not sufficient. It overwrites the rotated characters which is not really useful. As in the previous example, the standard defaults are reset before we end the program.

So much for text output. Circles and text do not constitute graphics. Lines must also be drawn, preferably in various thicknesses. You may also need to draw pointers and arrows in technical drawings. But they're no problem for the VDI.

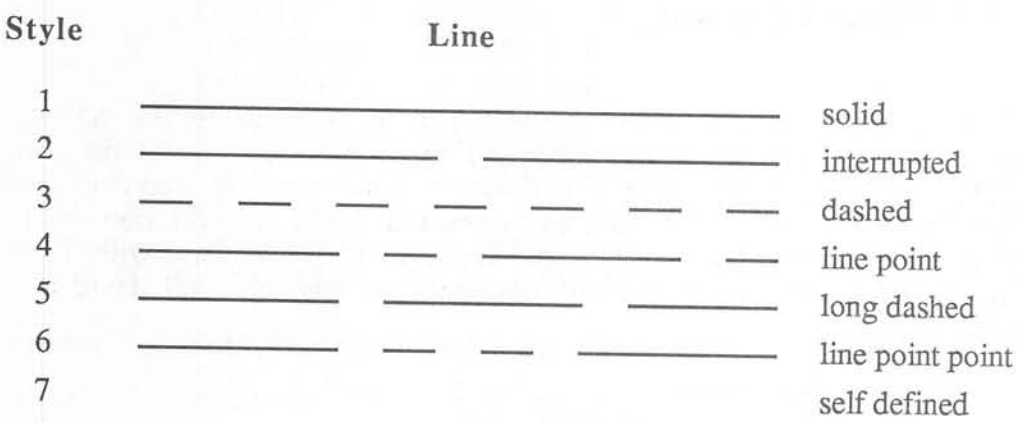
7.1.3 Lines and arrows

Drawing a line in the output window can be done using the LINEF command, where the starting and ending coordinates of the line are specified. The corresponding VDI function is number 6, named *Polyline*. Polyline is the name for several connected lines. Several coordinate points can be passed to the function. It simply connects them with lines. Simple drawings such as frames can be drawn with one call. Here is a sample program:

```
10   rem *** Multiple Lines 7.1.3***
20   clearw 2:fullw 2
30   poke contrl,6     : rem Polyline Code
40   poke contrl+2,4   : rem Number of Points
50   poke contrl+6,0
60   poke ptsin,50          : rem X 1
70   poke ptsin+2,50        : rem Y 1
80   poke ptsin+4,150       : rem X 2
90   poke ptsin+6,100       : rem Y 2
100  poke ptsin+8,160       : rem X 3
110  poke ptsin+10,180      : rem Y 3
120  poke ptsin+12,50       : rem X 4
130  poke ptsin+14,50       : rem Y 4
140  vdisys 0  : rem and normalize execution
```

This program draws a triangle on the screen. It was necessary to give the starting and ending point twice. The number of the coordinates are passed in CONTRL(1). There is a closely related function which not only draws a polygon, but also shades it. You can use it by first using a COLOR command to determine the shading attributes and using command code 9 instead of 6.

Let's continue with Polyline. It would be nice if we could draw lines as dots or dashes. Let's make this clear to the VDI. With function 15 you can select 7 different line types. This style is transmitted in INTIN(0), and means the following:



Style 7 can be user defined. The definition consists of a 16 bit word and its bit pattern determines the line. The solid line is defined as 1111111111111111 = $FFFF = 65535. This number is passed to function 13 and determines the form of every line which is designated afterwards with style 7. Here is a program which defines the two lines and draws them:

```
10    rem *** Set Line Style 7.1.3***
20    clearw 2           : rem Clear OUTPUT window
30    poke contrl,15     : rem Select Line style
40    poke contrl+2,0    : rem Number of Parameters
50    poke contrl+6,1
60    poke intin,7       : rem Line Style User
70    vdisys 0           : rem and execution
80    poke contrl,113    : rem define line
90    poke contrl+2,0    : rem Parameter number
100   poke contrl+6,1
110   poke intin,65365   : rem Line pattern
120   vdisys 0           : rem and execution
200   linef 0,20,200,20  : rem Trial line
```

The lines can also be drawn in variable thicknesses. Note that thicker lines can only be represented in style 1. If another style is selected, it is ignored. The thickness of the line can be set between 1 and 15 by calling function 16. An example:

```
10   rem *** Set Line Width 7.1.3***
20   clearw 2   : rem Clear OUTPUT window
30   poke contrl,16 : rem Select Line width
40   poke contrl+2,2 : rem Number of
     Parameters
50   poke contrl+6,0
60   poke ptsin,15          : rem Line Width
70   vdisys 0   : rem and normalize execution
100  linef 0,30,250,100   : rem Trial line
```

The shape of the line endings of these lines can also be defined. There are three selections possible:

0 normal angle
1 arrow
2 rounded

They are set with function 108, separately for the beginning and ending of the line. To draw an arrow from the position X1/Y1 to position X2/Y2 and to round off its beginning, our program would look like this:

```
10   rem *** Draw Arrow 7.1.3***
20   poke contrl,108      : rem Define end
30   poke contrl+2,0
40   poke contrl+6,2
50   poke intin,2    : rem Beginning rounded
60   poke intin+2,1 : rem End arrow shaped
70   vdisys 0   : rem and normalize execution
```

Then we draw the line as in the previous example. The rounding off of the line end is only noticeable with thick lines. Arrows can be drawn with any line width.

7.1.4 Shading surfaces

If you draw a form you might want to shade it. In BASIC the command for this would be `FILL X,Y`. The shading begins at the position given, and shades everything within an uninterrupted outline. The

shading and the color is set initially with the COLOR command. This command expects 5 parameters. The following table explains the parameters:

Parameter of COLOR A,B,C,D,E

A	Color of text
B	Background color of FILL command
C	Line color of drawings
D	Pattern index
E	Shading style

Pattern index and shading style specify the appearance of the shading. The shading style is the method in which the shading is performed:

Style	Shading
0	Area will not be shaded
1	Area will be completely shaded
2	Shaded with dots
3	Shaded with lines
4	Shading with self defined pattern

The pattern index is only important with style 2 and 3 and indicates which dot or line pattern will actually be used.

7.1.5 Creating your own shading patterns

Shading style 4 is very interesting. This allows you to define your own shading pattern. You can define a logo or a special character for a pattern. Any pattern which can be designed within a 16x16 pixel area may be defined. The program is written so that you can insert your own pattern. Let's design a pattern based on the little man from an earlier example.

```
10    rem *** Define Shading Pattern 7.1.5***
20    poke contrl,112      : rem Command code
30    poke contrl+6,16: rem Number of
      parameters
80    for i=0 to 15 : read x$   : rem Shading
      pattern
82    x=0 : for j=1 to 16 : rem change
84    x= x-(mid$(x$,j,1)<>" ") * 2^(16-j)
86    next j
90    poke intin +i*2,x    : rem set pattern
100   next i
110   vdisys 0   : rem and normalize execution
112   color 1,1,1,1,4 : rem select pattern
114   pcircle 80,80,70: rem and demonstration
120   end
130   rem +++Sample Data +++
140   data "                "
150   data "       ***      "
160   data " **   * * *     "
170   data "  **   ***      "
180   data "    **   *      "
190   data "      *******    "
200   data "      ***** **   "
210   data "      *****  *   "
220   data "      ***** **   "
230   data "      *****      "
240   data "      ** **      "
250   data "      ** **      "
260   data "      ** **      "
270   data "      ** **      "
280   data "    **** ****    "
290   data "                "
```

To draw this or any other user-defined pattern on the screen, call the VDI using function 103. This is the same function that is used by the `FILL` command of BASIC. The method is the same as in `FILL` concerning the attribute setting and the method of shading procedure. A VDI call for shading looks like this:

```
10    rem *** Shading Demo 7.1.5***
15    color 1,1,1,3,2 : rem Shading Attribute
20    poke contrl ,103 : rem Shading command
30    poke contrl+2,2  : rem Parameter Count
40    poke contrl+6,1
50    poke intin  ,1     : rem Shading color
60    poke ptsin  ,100   : rem X Coordinate
70    poke ptsin+2 ,200 : rem Y Coordinate
80    vdisys 0  : rem and normalize execution
90    pcircle 80,80,70
```

Incidentally, you can add dots or lines to the menu bar of GEM if you wish.

7.1.6 Setting markers in the display

In some programs you may want to draw markers or special symbols on the screen, perhaps to indicate a certain position. You can do this by drawing small circles using a CIRCLE command or you can use the following VDI function to draw one of several built in markers. The various types are:

Type	Form
1	Period
2	Plus sign
3	Asterisk
4	Square
5	Cross
6	Diamond

Any other value will produce an asterisk. Except for the period, the symbols can be enlarged—to provide any size you desire. For example, you can frame a character within a square.

The selection of type, size, shading color, and symbol are done individually. This makes for more work, but their flexibility makes it worth the effort. Many programs use these markers to indicate menu choices in dialog boxes.

Let's examine a program where all the selections are made and several markers are set:

```
10    rem *** Poly Marker 7.1.6***
15    clearw 2:fullw 2
20    poke contrl,18        : rem Shading code
30    poke contrl+2,0       : rem Parameters
40    poke contrl+6,1
50    poke intin,6          : rem Type diamond
60    vdisys 0  : rem and normalize execution
70    poke contrl,19        : rem Size
80    poke contrl+2,1
90    poke contrl+6,0
100   poke ptsin,0          : rem Dummy
110   poke ptsin+2,30       : rem Marker size
120   vdisys 0
130   poke contrl,20        : rem Set Color
140   poke contrl+2,0
150   poke contrl+4,1
160   poke intin,2          : rem Color number
170   vdisys 0              : rem set
180   poke contrl,7         : rem Polymarker
190   poke contrl+2,2       : rem Point number
200   poke contrl+6,0
210   poke ptsin,50         : rem X 1
220   poke ptsin+2,50       : rem Y 1
230   poke ptsin+4,150      : rem X 2
240   poke ptsin+6,100      : rem Y 2
250   vdisys 0              : rem and draw
```

Function 7, set marker, simultaneously draws all symbols whose coordinates were passed to the function. These coordinates `Xn/Yn` can be anywhere on the screen. However, all subsequent calls to this function use the original attributes again.

7.1.7 Testing points on the screen

VDI function 105 can determine the color of a point on the screen. If the color is the same as the color of the background, this condition is reported. This information is passed back in array elements INTOUT(0) and INTOUT(1). With this information you can determine if an object collided with another in a game, for example. You could also use this information to copy a portion of a picture. A program for examining a point on the screen is as follows:

```
10    rem *** Sense Point on Screen 7.1.7***
20    poke contrl,105      : rem Command Code
30    poke contrl+2,2: rem Parameter Count
40    poke ptsin,x         : rem X Coordinate
50    poke ptsin+2,y       : rem Y Coordinate
70    vdisys 0             : rem and test
80    set = peek(intout): rem 1 = set/0 = not
90    colour = peek(intout+2): rem 0/1 with
      monochrome
```

7.1.8 Mixing colors

You can set 16 different colors with the control panel, which can be displayed simultaneously on the color monitor. These colors are later specified by selecting a number from 0 to 15 with the COLOR command. Sometimes you may want to change the color in a program being executed. This can't be done by selecting colors on the control panel. BASIC doesn't have the capability to manipulate the individual color registers, so this must be done through the VDI.

VDI function 14 permits the exact selection of a color register. This is done by mixing 3 additive color portions where the intensity of the basic color is selected individually. The gradation ranges from 0 to 1,000.

The three intensity values are passed to the function simultaneously by calling the VDI. Items already drawn in the selected color immediately take on the new color since even with tricks, the ST can't produce more than 16 colors!

Here is a sample program to set the color number 2 to a brown tone. This program only works on a color monitor:

```
10   rem *** Set Color 7.1.8 ***
20   poke contrl,14    : rem Function Number
30   poke contrl+2,0
40   poke contrl+6,4
50   poke intin,2          : rem Color Number
60   poke intin+2,600      : rem Red portion
70   poke intin+4,400      : rem Green portion
80   poke intin+6,200      : rem Blue Portion
90   vdisys 0   : rem and normalize execution
```

To set the exact color you want with these red, green and blue portions, you have to experiment. This can be fun since you can discover the great color capabilities of the ST.

7.2 Music and sound

The ST has three built-in tone generators whose sounds can be mixed. In addition, every tone channel can be used as noise generator. It also has a sine wave generator, which lets you create some very interesting sound effects.

You can set tone variations with a combination of SOUND and WAVE commands. These commands offer a wide range of possibilities for representing realistic sounds.

The simplest way to create a tone is to push a key. The click generated by this is a tone generated by the sound chip. This explains why you can interrupt a tone by pushing a key. The operating system contains routines to create the clicking sound. Because of this some of the tone variations are reserved for the operating system.

The keyboard click and the bell which sounds during certain error messages are generated by an interrupt routine. Each 1/60th of a second (1/50th on PAL systems), the interrupt routine is called to determine the value of a two-byte pointer into a special table. If the value of the first byte of the pointer is zero, the data contained in this table is used to program the sound chip. If the value of the first byte of that pointer is $FF, normal processing takes place.

To use one of the preset tones from the operating system we set the pointer located at $E44 to the location of the sound data for the click:

```
10 dedfdbl a
20 a = 3652
30 poke a,36612
```

To produce a bell tone, we set the pointer to a different set of sound data:

```
30 poke a, 36642
```

We can also produce a sound of our own by pointing to our own sound data. Before doing this, let's look at the sound chip a little closer.

The values from the sound data table are passed to the sound chip registers in the I/O area of memory.

In early versions of GEM, this I/O area was protected, so that only privileged programs were allowed to access these ranges. To "bypass" the protection we can write the following machine language routine which temporarily puts the user's program into supervisor mode:

Source File: BYPASS.S

```
1           * Bypass privileged access *
2
3   000000 B07C0002     CMP.W #2,4(A7)      * 2 arguments ?
4   000004 661C         BNE  EXIT           * no => Exit
5
6   000006 42A7         CLR.L -(SP)
7   000008 3F3C0020     MOVE.W #$20,-(SP)
8   00000C 4E41         TRAP  #1            * Supervisor 1
9   00000E 5C8F         ADD.L #6,SP         * Stack rep.
10
11  000010 226F000E     MOVE.L 14(SP),A1    * Get address
12  000014 32AF0012     MOVE.W 20(SP),(A1)  * and set
13
14  000018 2F00         MOVE.L D0,-(SP)
15  00001A 3F3C0020     MOVE.W #$20,-(SP)
16  00001E 4E41         TRAP #1             * User State
17  000020 5C8F         ADD.L #6,SP         * Stack rep.
18
19  000022 4E75         EXIT: RTS           * and back !
```

To use this routine from BASIC, we write a simple loader which puts the machine code into the memory reserved for a BASIC variable called A$.

```
10   rem *** Privileged Access 6.4.1***
20   a$=space$(36)          : rem reserve space
30   b=varptr(a$)    : rem determine address
35   b=b+(b mod 2)   : rem set even address
40   for i=0 to 34 step 2
50   read a                 : rem read data
60   poke b+i,a             : rem and write
80   next i
```

```
90    rem ---Machine Program - Data ---
100   data -20356,2,26140,17063,16188
110   data 32,20033,23695,8815,14,12975
120   data 20,12032,16188,32,20033,23695,
      20085
200   call b(16746496,8*256): rem register
210   call b(16746498,12*256): rem value
220   end
```

The FOR/NEXT loop in lines 40-80 POKE the machine code into variable A$. Then the CALL statement in line 200 performs the equivalent of a POKE but bypasses the protection normally afforded the I/O register memory area.

The routine first checks to make sure that two parameters are being passed from BASIC. If not, control is returned back to BASIC.

If two parameters are passed from BASIC, the TRAP instruction sets the program to supervisory mode and the two arguments may be used as privileged POKE equivalents. Finally a second TRAP instruction resets the mode back to unprivileged access.

Now we can program the sound chip by replacing statements 200-220 with the following statements:

```
199   rem*** noise and music 7.2***
200   read r,w          : rem Register and Value
205   if r=-1 then end     : rem Test for end
210   call b (16746496,r*256) : rem Select
      register
215   call b (16746498,w*256) : rem write
      value
220   goto 200               : rem continue...
249   rem              Switch on A and B
250   data 7,252
259   rem              Channel A Loudness
260   data 8,12
269   rem    Channel A on Sine wave,Frequency
270   data 9,16,3,2
279   rem       Frequency of Sine wave
280   data 13,10
300   data -1,0:rem END
```

This program creates a tone from two frequencies where one is varied through a sine wave. The data pair starting in line 250 is a register number of the sound chip and the value which is to be written to this register. We can now vary the register values to hear the effect of different combinations. The register and the corresponding content have different uses which we'll look at now.

Register Number	Significance
0 and 1	Duration of period of channel A. A total of 12 bits in this word are used.
2 and 3	Same, only for channel B.
4 and 5	Same, only for channel C.
6	Duration of period for noise generator with the lower 5 bits used.
7	Configuration register. Every bit has its own assignment: Bit 0: channel A 0=on, 1=off Bit 1: channel B 0=on, 1=off Bit 2: channel C 0=on, 1=off Bit 3 channel A with noise 0=yes, 1=no Bit 4 channel B with noise 0=yes, 1=no Bit 5 channel C with noise 0=yes, 1=no Bit 6: Port A data direction 0 = in, 1 = out Bit 7: Port B data direction 0 = in, 1 = out
8	Volume of channel A. The lower 4 Bits are valid. If bit 4 is set, the Sine wave is turned on and the lower 4 bits are ignored.
9	Same as above, but for channel B.
10	Same as above, but for channel C
11 and 12	Duration of period of sine wave (LO,HI). All 16 bits are used.
13	Sine wave curve (Bits 0 to 3).
14	Port A data register
15	Port B data register

The two ports, A and B are programmable data registers which also have some assignments that are really are not related to tone generation. Port B is directly connected to and controls the parallel port of the printer, while port A controls the selection of the disk drive, the data request of the serial interface and the GPO (General Purpose Output) connection of the monitor connector. Changing the contents of these registers is therefore not recommended. Let's stick with the tone generation.

By experimenting enough, you will find a combination of values that suits your musical tastes. If we create a table for the registers and the desired content, and store them in memory, change the pointer of the interrupt routine to access the data in our table. The tone or a series of tones can be played while the BASIC program continues to execute. You can play background music or, as in arcade games, create the noise of a shot or hit while the game continues.

This technique of generating sound is not limited to tone generation. While the music is being played, you can generate pauses which are related only to the tone generation and not the BASIC program. Furthermore, rising or falling values can be programmed so that you can create a siren sound with little effort. Let's consider a sample program which produces a typical falling noise:

```
10   rem ** Create Tone Sequence 7.2**
20   def seg=0
30   defdbl b : b=3652    : rem Set long word
40   a$=space$(100)       : rem Create space
50   a=varptr(a$) : rem Determine
     address
60   def seg=1
70   for i=0 to 100
80   read x          : rem Pass values
90   poke a+i-1,x
100  if x=255 then 120
110  next i
120  poke a+i,0      : rem Conclusion zero
130  def seg=0
140  poke b,a        : rem and start tone!
```

```
150  rem --- Tone Data ---
160  data 0,1,1,0,2,0,3,0
170  data 4,0,5,0,6,0,7,254
180  data 8,16,9,0,10,0,11,0
190  data 12,35,13,10
195  data 128,50
200  data 129,0,1,250
205  data 200,30
210  data 0,0,1,2,2,0,3,0
220  data 4,0,5,0,6,0,7,246
230  data 8,16,9,0,10,0,11,0
240  data 12,62,13,9
250  data 255
```

The first block of data in lines 160 to 190 creates a tone whose volume is modulated with a triangular curve. This is accomplished by setting the volume on channel A in register 8 to 16, which creates the dependence on the sine wave. The sine wave itself is set in registers 11,12, and 13 which can be seen in the table of the sound register chip.

Now to lines 195 and 200. Here two special commands are issued for the sound interrupt. These special commands are used instead of registers and are always set larger than 127. The special command `128` causes the following value to be stored (nothing else). The `129` starts the following sequence:

The number after the `129` is interpreted as a register for the following data. This register is first selected, then the value stored previously with the `128` command is stored in it. In our example, the period duration of channel A and its frequency is set. A tone starts which varies in loudness according to the sine wave.

The constant following the register value is constantly added to this register. This leads to the constant decrease of the pitch, since an increasing value causes a decreasing frequency.

The last number is the final value which the register can contain. In our example this means the lowest tone which channel A can reach.

If the timing of the example above is set properly, the tone is at the lowest frequency when the sine wave has reached its maximum. If you imagine a falling body, it must strike the ground at this point. To make

things more interesting, a small pause is introduced before this occurs. The third special command causes this pause. This must be some value above 129. The number following sets the duration of the pause. During this time period nothing is changed in the condition prevailing in the sound chip register and therefore the tone remains the same. A delay can be set with this command and the WAVE command before a tone sounds.

With this trick of tone generation through interrupt routines you can create quite a few noises to use in a game. However, you should be aware that pressing a key will alter the pointer again and turn off the tone. This is also true of noises which were set with SOUND and WAVE. These two commands will be examined next.

The SOUND command can accept 4 or 5 parameters. These values have the following meaning with the command SOUND A,B,C,D,E:

A	Channel number (0-2)
B	Loudness (0-15)
C	Musical note of the tone (1-12)
D	Octave of tone (1-8)
E	Tone duration (0-255 in 1/50 seconds) can be omitted

The frequency setting is also divided into octaves and notes which makes the translation of a musical piece to BASIC, according to notes, very easy.

The WAVE command also requires 4 to 5 parameters. Here WAVE A,B,C,D,E means the following:

A	Configure. Here the register 7 of the sound chip is set. However, only the lower 6 bits are used.
B	Sine wave switch on/off
C	Set sine wave, register 13
D	Period duration of the sine wave
E	Delay of the tone (can be omitted)

With these BASIC commands you can try to create the same falling noise in the above program. It is amazing how many different noises can be coaxed from the ST!

7.3 Window and Menu programming

What distinguishes the ST from most other computers is its user-friendliness, since you work with menus and can configure a desk to suit your own preferences. You can also use this technique in your programs.

The simplest method to build a menu is to display several selections with code numbers and to write them to the screen. The user is then requested to enter the number of his choice. The program then runs the subroutine which was selected. The whole thing appears as follows:

```
5     rem*** datainpt.bas 7.3 ***
10    print "1) Input Data"
20    print "2) Print Data"
30    print "3) End"
40    input "Please select ";W
60    on W goto 100,200,80
70    goto 40
80    end
100   rem ** Data Input **
110   .....
200   rem ** Print Data **
210   .....
```

This type of menu operation is used in many programs. However, the ST offers the capability to make this more convenient.

The ST has 10 function keys. You can use these function keys in a similar way in your programs, but they are easier to handle than the above method.

You can eliminate the need to press the <Return> key by adding:

```
50    w=inp(2)-186
```

Since the F1 key produces a value of 187, we can sit variable `W` to 1 by subtracting 186. We then use this value with the `ON...GOTO` command to branch to the right section. This is the first step in simplifying the menu. This technique is also used in many programs.

But let's go to the next step towards a better menu. GEM offers several features for easy menu operation. We have already used the VDISYS command for programming with GEM. The VDI helps us very little here, since it is only responsible for graphics. We shall now examine the mysterious GEMSYS command.

GEMSYS activates GEM in the ST, just like VDISYS. The difference lies in the fact that VDISYS only calls VDI, while GEMSYS is responsible for the AES. The AES is responsible for the windows and menu processing in the ST. It permits the user input with the mouse, for example.

Just like the VDISYS command, GEMSYS requires parameters to know what it must do. These parameters are passed to the AES in arrays which have the names CONTRL, INTIN, INTOUT, ADDRIN, ADDROUT. You might think that you know the first three! Wrong! There is a big problem. These CONTRL, INTIN and INTOUT arrays are not the same ones that we used with the VDI! These data fields are also located in different areas of the memory.

The question is how to access these addresses. The systems variables of BASIC for CONTRL and the others can't be used and ADDRIN or ADDROUT are completely unknown to BASIC. What it does understand and what we haven't discussed up to now, is the system variable GB. This is a pointer just like the others. It points not to a data field, but a table where other pointers are stored. And these are the pointers we need.

These are 32 bit pointers whose sequence is as follows according to their use:

```
PEEK(GB)        CONTRL
PEEK(GB+4)      GLOBAL
PEEK(GB+8)      INTIN
PEEK(GB+12)     INTOUT
PEEK(GB+16)     ADDRIN
PEEK(GB+20)     ADDROUT
```

These pointers are used for parameter passing to AES, i.e. the GEMSYS command. The GLOBAL array plays a subordinate role. It contains information on the status of the GEM-AES commands currently in execution. Here are the meanings of the entries in the GLOBAL field:

GLOBAL	GEM-Version Number
GLOBAL+2	Maximum number of simultaneously active programs
GLOBAL+4	Number of actual programs
GLOBAL+10	Pointer to a tree structure

We can usually ignore this field. The other arrays are more important. The CONTRL field has the same partitioning as the VDI. CONTRL(0) contains the function number, CONTRL(1) and CONTRL(2) the number of INTIN and INTOUT entries and CONTRL(3) and CONTRL(4) the number of entries in ADDRIN and ADDROUT fields in long words.

Let's begin using the GEMSYS commands with a simple call. The following example is the function FORM_ERROR which displays a TOS error message in a small window in the middle of the display. The error number is passed through INTIN(0).

```
10   REM *** TOS-ERROR-DEMO 7.3***
20   DEFDBL B : B=GB
30   CN =PEEK(B)      : REM POINTER TO CONTRL
40   II =PEEK(B+8)    : REM POINTER TO INTIN
50   POKE CN,53       : REM FUNCTION NUMBER
60   POKE CN+2,1
70   POKE CN+4,1
80   POKE CN+6,0
90   POKE CN+8,0
100  POKE II,22             : REM ERROR CODE
120  GEMSYS 53              : REM AND RUN
```

Let's look at this example more closely. First of all, you will note that we don't work directly with GB, instead we access it through variable B. The reason for this is that the pointers CN and II are longwords (4 bytes long). A PEEK(GB) only produces the HI word of the pointer CONTRL which is useless. Since the variable B is defined as double precision, the PEEK(B) results in a long word.

Next you might ask why CONTRL and INTIN were not used as variable names which would make the program easier to read. Unfortunately, these two names are reserved for the system variables so they may not be used for other variables.

Another difference from the VDI programming is evident here. The GEMSYS command is written directly with the function number, the value following GEMSYS is no dummy but the function number itself.

Showing a TOS-ERROR message is not very useful for a BASIC program. It would be more interesting to show some other text and perhaps to create some menus. This is also possible by using the AES function FORM_ALERT, which is provided for error messages but could be used for other applications. Two parameters are passed to the function. The first of these is a pointer to the text, which represents the information, the address of the (3 maximum) selection points and the number of the symbols to be displayed near the message.

The text can simply be stored in a text variable and the we can pass the address of the variable to the AES. Furthermore, another parameter is passed which determines which of the selection choices can be selected with the <Return> key. It will be darkly framed similar to the OK choice in many dialog boxes.

In the early versions of BASIC there is another problem. The value of the selection is returned in INTOUT(0). Before the value can be read with PEEK(IO) it is changed again by another call from BASIC. Up to now we have not been successful in obtaining the correct value. The only solution is to pass a 0 in INTIN(0). By doing this no selection can be chosen with the <Return> key. The selection can be made only with the mouse;its position can be determined after the return from AES. The Y coordinate of the mouse pointer can be ignored since the selection points are usually adjacent to one another.

The function call with the text necessary is shown in the following program:

```
10   rem *** Alert - Demo 7.3 ***
11   defdbl b,d : b=gb
12   cn=peek(b)      : rem Define pointer
14   ii=peek(b+8)
20   io=peek(b+12)
25   d =peek(b+16)
30   a$ = "[1][You have the choice: ......]"
31   a$=a$+"[Key 1|Key 2|Key 3]"
40   a = peek(varptr(a$)+2)
50   poke cn,52           : rem Function code
```

```
60   poke cn+2,1
70   poke cn+4,1
80   poke cn+6,1
90   poke cn+8,0
100  poke ii,0    : rem No 'Return' selection
110  poke d,a       : rem Text address
120  gemsys 52      : rem Execution
125  t=peek(io)     : rem Actual key number
```

The [1] stands for the symbol which appear near the text of the message. You have a choice between a STOP sign, a question mark, or an exclamation point. The second bracket contains the text of the message. Line feeds are made with the vertical line. The length of the text should be selected in such a manner that the window with the selection choices is proportioned properly. These choices are defined in the contents of the third bracket in the text. You can have a maximum of 3 choices with a maximum of 20 characters in length. The separation of the individual entries is again marked by the vertical line.

We have already found several ways to determine the position of the mouse. Since we are discussing the AES, we'll use a GEMSYS command.

This function returns more than the position of the mouse pointer. You also obtain the condition of the two mouse buttons and the <Shift>, <Control> and <Alternate> keys. The call requires no parameters.

```
10   rem *** Mouse Status 7.3 ***
11   defdbl b : b=gb
12   cn=peek(b)
25   io=peek(b+12)
50   poke cn,79          : rem Function code
60   poke cn+2,0
70   poke cn+4,5
80   poke cn+6,0
90   poke cn+8,0
100  gemsys 79           : rem call of AES
110  for i=2 to 8 step 2
120  x(i)=peek(io+i)     : next i
130  print "Position  ";x(2);" , ";x(4)
140  print "Mouse Key ";x(6)
150  print "Key     ";x(8)
```

We store the values in variable array X, since a PRINT command, for example, would directly output the position of the mouse. This would call the AES and change the parameter table. It is interesting to note that after the call of function 52 the INTOUT field is destroyed with this same function 79. Determining the mouse position can be performed without any other GEMSYS commands.

The AES functions presented above used the output window on the screen. In addition, BASIC offers four windows whose position and size may be changed using the mouse pointer. To change one of these windows from inside a program, another AES call is required. It concerns the function 105 which has the name WIND_SET. With this function you can move the window and change the outside edge of the window. For example, it can prevent that window from being closed with the mouse.

Here a sample program to set the position and size of a LIST window. By inserting another number in line 40, one of the other windows can be changed.

```
10   rem *** Set Window 7.3 ***
20   defdbl b : b=gb
30   ii=peek(b+8)
40   h=2              : rem LIST window
45   openw h-1        : rem open window
50   poke ii,h        : rem select
60   poke ii+2,5      : rem Mode
70   poke ii+4,20     : rem X coordinate
80   poke ii+6,20     : rem Y coordinate
90   poke ii+8,350    : rem Width of window
100  poke ii+10,100 : rem Height
110  gemsys 105       : rem and set
```

The coordinates give the position of the upper left corner of the window. The mode 5 is passed in line 60 sets the function to be executed by AES. The following modes are permitted:

Mode 1:	in `ii+4` a number is expected whose bit combination defines the window frame area. The bits signify the following: Bit 0: Title line of the window Bit 1: Delete field Bit 2: Opening field Bit 3: Movement field Bit 4: Information line Bit 5: Size change field Bit 6: Arrow up Bit 7: Arrow down Bit 8: Vertical mover Bit 9: Arrow left Bit 10: Arrow right Bit 11: Horizontal mover
Mode 2	The name of the window is changed. The address of the new name is expected in `ii+4` to `ii+7` as longwords. The text must be concluded with a zero.
Mode 3	The information line of the window is changed. The conditions the same as in mode 2.
Mode 5	The window is set. This mode will be used in our example.
Mode 8 and 9	The relative position of the horizontal or vertical shifter is set.
Mode 10	The actual window is selected.
Mode 15 and 16	The relative size of the horizontal or vertical shifter is set.

With these functions any window adjusted. BASIC program are made more flexible this way.

7.4 Text processing

Text plays an important role in computer applications. A computer can do more than simply calculate. It can also manage, change, store, and print text. But text, in our case string variables, can be used for other purposes. You can store other data such as machine language programs in text variables. We have used this method before.

To store a text variable in a storage area, you can work with the variable pointer VARPTR() and POKE. VARPTR(A$) points to a table in which some information about the variable A$ is stored. The third and fourth bytes are the storage address of the variable itself. It can be found with the command A=VARPTR(A$). To load a file from diskette into the string variable, which is also called a string, you can use the following program:

```
10    rem *** Store File in String 7.4 ***
15    a$=spaces$(200)    : rem Reserve Spaces
17    a=varptr(a$)       : rem Determine Address
20    input "Filename ";fn$ : rem Input File Name
30    bload fn$,a        : rem and read in
40    print :o=(peek(a) and 255) : rem Calculate
      offset
50    e=o+peek(a+4)             : rem calculate end
60    for i=0+2 to e step 2
70    print i ,peek(a+i)        : rem Display content
80    next i
```

With this program small files on disk (less than 200 bytes) can be read into A$. Lines 40 to 80 are not required, but they are meaningful in connection with an applications program (*.PRG). If one is loaded, the beginning and the end of the actual program are calculated, and the values output as decimal words. You can use this program to convert the small machine language programs created with the assembler into DATA lines. To do this you only need to type in the value output in DATA lines, and to read it with the READ-POKE loop into the string again.

You can use this technique to store a larger text field to a diskette. It can also be done with OPEN and PRINT#, but that takes considerably longer than BLOAD or BSAVE.

Now we don't want to forget the real reason for text variables and their usage. One of them is to store inputs, and if necessary to sort them. A sort program of this type is very simple with the ST BASIC since it has a SWAP command. Here is an example using the bubble sort algorithm:

```
10   rem *** Sorting of Text 7.4 ***
20   dim w$ (10)
30   for x=1 to 10
40   print x;". Word :";input w$ (x)
50   next x
60   print "Sorting in progress"
70   for i=1 to 10
80   for j=i to 10
90   if w$ (j)<w$ (i) then swap w$(i),w$(j)
100  next j
110  next i
120  for x=1 to 10: print w$(x) :next x
```

First, you are asked to enter ten words. These are stored in the text field W$ (). Next, they are sorted by comparing the words during a loop and if necessary, their order is exchanged with the SWAP command.

The text in the above program can be names, addresses or telephone numbers which are stored in a file. This sort routine can also be used for larger text fields, such as multi-dimensional string arrays. The desktop can sort the table of contents for a disk according to various criteria, such as name, type or length. A BASIC program could sort the addresses and output them according to name, address or phone number. The input for the data would appear nicer if they were arranged with a template.

7.4.1 Templates

Templates are used in most common database programs. A template is shown on the screen in which space has been left for entries. The user can enter the data only in the space made available. The completed screen will look just like the template, which is then printed. A template is also referred to as a screen mask.

```
10    rem *** Template 7.4.1 ***
15    res = peek(systab)
20    dim t$(100,6),x(7),y(7) : rem Define fields
30    fullw 2 : clearw 2      : rem Prepare window
40    for i=1 to 7
50    read x(i),y(i),a$       : rem Read mask
60    gotoxy x(i)*res , y(i)/res  : rem Set cursor
70    print a$;left$(".........",11- len(a$));" :"
80    next i
90    data 10,1,"** Input Address **"
100   data 1,5,"Last Name"
110   data 16,5, "First Name"
120   data 1,10, "City"
130   data 1,12,"Street"
140   data 1,15,"Telephone"
150   data 2,18,"Remarks"
160   n=1                     : rem First entry
165   gotoxy 1,1 :print n;" )" :rem Write number
170   for i=1 to 6
180   gotoxy x(i+1)*res+6*res,y(i+1)/res:rem cursor
190   input t$(n,i)                 : rem Input
195   if t$(n,i)="#" then i=10 : rem Termination
200   next i
210   if i<10 then n=n+1 : goto 165
220   n=n-1                   : rem Correct number
```

In this program a template is constructed and displayed on the screen, in the output window. After that the input is made. If a # is entered somewhere, input is terminated. In the following lines the input is processed, where `T$(n,1)` contains the nth. last name and `T$(n,2)` contains the nth. first name etc. The variable `N` contains the number of entries which were made.

Line 15 checks the screen resolution, which is used in lines 180 and 60 so the program will run on either a mono or color system.

Now the sort routine can be employed. If you want to sort the phone numbers, only `T$(n,5)` must be sorted.

7.5 Mouse/Joystick Control

The keyboard is generally used to control a program. For some applications it can be a burden to use the keyboard for a single input. This is the case with a game if it is played with a joystick. For a question such as if another game is to be played, it would be nice if input could also be done with the joystick. Such control is not difficult to program. A yes/no question can be answered with the right/left movement of the joystick or the pushing of the right or left mouse buttons. Of course, it is more complicated if several choices exist. For this situation we can display a menu where we can make choices with the mouse or the joystick. Here is an example of such a selection technique:

```
10    rem ** Selection with Joystick 7.5 **
15    def seg=1          : rem PEEK/POKE Byte
20    clearw 2 :gotoxy 0,1:rem Prepare window
30    print "  1.":print "   2.":print "   3."
31    rem Display Menu
40    y=peek(3592)       : rem Vertical joystick
45    poke 3592,0              : rem Reset
50    w=w-(y=1)+(y>1)-(w>3)+(w<1) : rem New position
60    gotoxy 0,w : print "=>"       : rem Show choice
70    for i=1 to 100 : next i : rem Pause
80    gotoxy 0,w : print "  " : rem Erase pointer
90    if (peek(3581)and 3)=0 then 40
```

In this program a pointer is pushed up and down between three selection choices and acknowledged with the fire button. The variable `W` contains the number of the selected menu choice. Of course any number of choices can be displayed if they can fit on the screen. Here is the version for the mouse:

```
10    rem ** Selection with Mouse **
15    res = peek(systab)
20    clearw 2 :gotoxy 0,1: rem Prepare window
30    print "  1.":print "   2.":print "   3."
31    rem Display Menu
40    y=peek(9954)              : rem Mouse position
50    w=1-(y>70/res)-(y>90/res): rem Determine choice
60    gotoxy 0,w : print "=>"       : rem Show choice
70    for i=1 to 100 : next i : rem Pause
80    gotoxy 0,w : print "  " : rem Erase pointer
90    if (peek(3581)and 3)=0 then 40
```

This program calculates the selected position for the arrow from the vertical position of the mouse pointer which is acknowledged with one of the two mouse buttons.

A selection arrow does not need to be shown since the mouse pointer is sufficient. The mouse pointer is more secure since small inconsistencies could produce false results.

The position of the mouse pointer can also be evaluated in two dimensions. The horizontal position of the mouse is, under normal conditions, available in storage location 9952, and the vertical in location 9954. These two storage words can be manipulated with `POKE` and set the mouse pointer to any desired position on the display. To bring the mouse pointer to the upper left corner of the display you only have to type `POKE 9952,0` and `POKE 9954,0`. This is handy when you have a design to do with the mouse and you want to define a starting position.

In line 15 the resolution is checked and is used in line 50 to allow the program to run on both the mono and color monitor.

7.6 Input/Output

Up to now we have relied mainly on storing data in the computer and displaying it on the screen. We shall now make the ST perform with external peripherals. For this a disk drive and printer are used.

The normal input and output to these devices has already been discussed, although the complete capabilities of the peripherals have not been exhausted. Disks are divided into sectors and access to individual sectors is not possible. The printer also has functions which we want to master. Finally, we want to discuss the operation and programming of a modem.

We already know the connections of the individual interfaces. We can now look at the programming of the individual peripheral units which are connected and perhaps create other connections.

7.6.1 Printer Control

The `LPRINT` or `LLIST` commands can be used to print out data and programs. In addition to these we have the capability to output with the `OUT 0,x` command with which any character can be output. This is not limited to the output of visible characters, but control characters can also be sent to the printer. The command `OUT 0,x` really corresponds to the command `LPRINT CHR$(x)` but is simpler to write and is faster.

Problems arise when different printers are connected to the ST. There is no printer standard concerning the control codes that enable a printer to print text and graphics. If you press the <Alternate>/<Help> keys the computer will send some control characters to the printer that disable the line feeds and set it into graphic mode. The problem which can occur is the incompatibility of some printers. This means that some printers have other control characters for various functions. The printer set-up which was prepared by the desktop is no help here. The operating system loaded from the disk, takes the required printer control characters from a table which is located in memory at `$16D5E`. This table can be manipulated to permit use of another printer with the ST. The sequence of the entries in the table are as follows:

$16D5E	<Esc> "X" 6	
$16D63	<Esc> "X" 5	
$16D68	<Esc> "X" 3	for Atari Color-Matrix Printers
$16D6D	<Esc> "X" 6	
$16D5E	<Esc> "L"	B/W Matrix Printer: 960 Dots/Line
$16D71	<Esc> "Y"	Color Printer: 960 Dots/Line
$16D75	<Esc> "3" 1	1/216 Inch Line Spacing
$16D7A	<Esc> "3" 1	See above
$16D7F	<Esc> "1"	7/72 Inch Line Spacing
$16D83	<Esc> "2"	1/6 Inch Line Spacing

For example, <Esc> L sets the Epson black/white printer to graphics mode. This character is followed by the number of bytes to be sent for the line. After that the bit pattern will print every additional byte in the actual line. If your printer has a different setting from the table above, you can adjust these with a `POKE` of the new characters.

Printing out the screen can start either with a combined keypress, or with the selection of a menu choice `Print Screen` or with the command `POKE 1262,0`.

To program the printer to print special symbols you have to send the proper command sequences to the printer. You have the additional choice of the resolution in which the printer should operate. <Esc> L sets this to double density which is necessary for printing a high resolution screen. You can select the normal resolution with an Epson black and white printer with <Esc> K. A small sample program for the output of special characters:

```
10   rem ** Print Special Character 7.6.1**
20   for i=1 to 16
30   read a          : rem Read character
40   out 0,a         : rem and output
50   next i
60   lprint          : rem Done; Line feed
70   data 27,75,12,128
80   data 4,10,26,58,103,231
90   data 231,103,58,26,10,4
```

This example outputs a small UFO on the printer. The 27 is the code for "Esc", the 75 is for "K". The 12 is the number of picture data items. With a number larger than 255, the high byte of the number must be added to the 128. After that, follow the data whose lowest bit is printed below. See your printer manual for more information on programming your printer.

7.6.2 Using Disks

To use the disk drive for storing our program data, we can use the LOAD and SAVE commands, or the BLOAD and BSAVE commands which have already been described in section 2.2. The total control of disk access is under BASIC or GEMDOS. However, it is also possible to take control and to bypass the operating system. For this we need a machine language program that can be stored in string variables.

In this section we want to examine a program that can do more. The first part of the program reads the machine language program from DATA lines into the text variable. This technique has been used before so that no further explanation should be required. The second part of the program determines which sector of the disk to read. The value input is introduced into a machine language program to prepare it for this one sector. The string variables D$(0) and D$(1) are then prepared to recieve the data from the sector. Here two variables are needed which are located next to each other, since a string can only be a maximum of 255 bytes long. The variable D is then loaded with the beginning of the buffer.

Now follows the call of the machine language program which obtains the buffer address as a parameter. The disk drive starts and the contents of the selected sector are loaded into the buffer. Since each sector always contains 512 bytes, D$(0) and D$(1) are partially overwritten. Other data buffers can also be used, but you have to be sure the memory region is not used by some other program.

If the sector has been loaded into memory, its contents are displayed in hexadecimal and as ASCII characters. This is helpful if you want to investigate the directory of the disk. This table of contents starts in the Atari ST disk format on side 0 in track 1, sector 1.

```
10   rem *** Disk Access 7.6.2 ***
20   fullw 2    : rem Output window large
30   def seg=0       : rem Set word access
40   a$=space$(36)   : rem Prepare string
50   b=varptr(a$) : rem Determine Address
60   for i=0 to 40 step 2
70   read a               : rem Read Data
80   poke b+i,a           : rem and store
90   next i
95   rem --- Machine Language Data ---
100  data 8815,14,16188,1,16188,0,16188
110  data 1,16188,6,16188,0,17063,12041
120  data 16188,8,20046,-8196,20,20085
130  print :input "Load which Sector?";x
140  poke b+10,x     : rem Sector Number
142  poke b+14,y     : rem track
144  poke b+18,z     : rem side
150  dim d$(4)       : rem Prepare Buffer
160  d$(0)=space$(255)
170  d$(1)=space$(255)
180  d= varptr(d$(0)): rem Determine address
190  call b (d)           : rem Load Sector
200  def seg=1            : rem Byte Access
210  print :for i = 1 to 512
220  x% = peek(i+d-2)     : rem read Byte
230  if x%<16 then print "0"; :rem only for
     Format
240  print hex$(x%);" "; :rem write Hex Byte
245  if (i and 15)>0 then 280:rem Line done?
250  print " "; :for j = 17 to 2 step -1
260  x%=peek(i+d-j)
270  if x%<11 then ?"."; : goto 270
280  print chr$(x%);: rem ASCII Character
290  next j : print : rem next character
300  next i          : rem next line
```

The machine language program appears to be very simple:

```
MOVE.L 14(SP),a1        * Save Buffer address
MOVE.W #1,-(sp)         * Number of sectors=1
MOVE.W #0, -(SP)        * Unit 0
MOVE.W #1, -(SP)        * Track 1
MOVE.W #6, -(SP)        * Sector Number 6
CLR.L      -(sp         *
MOVE.L a1, -(SP)        * Buffer Address
MOVE.W #8, -(SP)        * Command floppy read
TRAP #14                * Execute
ADD.L #14,SP            * Repair Stack
RTS                     * Back to BASIC
```

Not only can the sector number be set, but also the unit number, the number of sectors to be loaded, and the direction of the data. It is possible to write from the buffer to the disk sector by changing the 0 to a 1 for the data direction. **Caution:** attempts to write to a wrong sector can completely destroy the disk. For example if you were to write over the diskette's table of contents. We recommend that you work with a diskette that does not contain important material.

This is a way to make a direct access to the diskette. The normal way is to choose a name under which the data is stored on the disk. You issue a LOAD"*path:name.type*" command and the program is loaded. The path indicates the drive to be used, and/or the designation of the subdirectories, and the category in which the file is located. The name is the description of the file itself and can be a maximum of 8 characters. The type (or extension) which appears at the end of the designation, consists of three characters and is an abbreviation of the file usage. This extension is arbitrary, but some of the type designations are reserved. PRG means that it is an executable machine language program, BAS is used for BASIC programs, etc.

These extensions help in selecting files or programs. If you search for a BASIC program on a disk, the command DIR *.BAS will list all of the available BASIC programs.

File selection works in similar manner when the LOAD command is chosen. A window is opened on the screen which can be subdivided into smaller windows. The top line contains the selection criteria and the path designations used to select the files available. If the line contains *.BAS

all programs with the extension .BAS are shown. The asterisk character is a wildcard that represents all characters that could appear. You can only change the selection line and perhaps show the programs from another disk drive.

The entire window with all its operating characteristics is controlled by the AES. We can select a file with very little effort for a BASIC program. For example, if we are writing a data base management program and want to load a data file from the disk. To call the function that draws the selection window and takes over the control we must prepare some parameters.

First of all we need two buffers for this. In the first the selection criterion which we discussed before is stored. The other contains a file name. It is not required, but the selection of OK or Discontinue will store the name of the selected file in the buffer. Along with the path definition of the other buffer, we now have the exact definition of the file available which we can now use for storing or loading our data. Here is a program that calls this function:

```
10   rem *** File-Select 7.6.2 ***
20   poke contrl,122           : rem call VDI
30   poke intin,0
40   vdisys 0          : rem switch Cursor on
50   defdbl b,d : b=gb
60   cn=peek(b)
70   d=peek(b+16)      : rem Field Definition
80   io=peek(b+12)
90   a$="\*.PRG"       : rem Define Path
100  for i=1 to 40 : a$=a$+chr$(0) : next i
110  for i=1 to 15 : b$=b$+chr$(0) : next i
120  a = varptr(a$): c = varptr(b$)
140  poke cn,90        : rem AES Command
150  poke cn+2,0
160  poke cn+4,2
170  poke cn+6,2
180  poke cn+8,0
190  poke d,a                : rem Set Path
200  poke d+4 ,c             : rem Name Buffer
210  gemsys 90               : rem Call
220  print "In ";a$
230  print b$;" selected": rem Output Result
```

The beginning of this program consists of a VDI call. This function turns on the mouse pointer, which disappears when a key is pressed. If you omit the VDI function, and start the program without moving the mouse, the cursor remains invisible during the entire execution of the function. That is somewhat of a handicap for program selection.

Along with this the pointers for the AES parameters are defined. The explanation of this process is in the chapter on menu and window programming. The two buffers for the file definition are also prepared. The two loops load these buffers with the value 0. This is necessary for the proper operation of the AES routine.

Next, the parameters are transferred into the AES parameter tables. `ADDRIN(D)` receives the two addresses of the buffers as longwords. Finally we call the AES. After clicking one of the two choices or a double click on a file name, the selection window is erased and control is passed to the BASIC program which issued the call. The two text variables now contain the path, or name of the selected file.

In addition, the function transfers which way (`OK` or `Cancel`) the termination of the function was achieved. The information is contained in `INTOUT(1)`. Unfortunately BASIC destroys this information before it can be read, so we must evaluate the position of the mouse.

7.6.3 Telecommunication

Telecommunication has developed into a widespread hobby among computer owners. In the United States an enormous number of public bulletin boards exist and choosing one of them can be a difficult job. Let's look at how we can join the world of communications with the ST.

First of all we need a telephone *modem*. This equipment, which connects the telephone and the computer, is available in many stores and through mail order companies. You should make sure to follow FCC regulations when you connect a modem to your phone line.

A commercial modem usually has a serial port for connection with the computer and a telephone jack which you connect to your telephone. The speed of the transmission can be set at several different speeds.

First we set the serial interface to 300 baud. Since most bulletin boards send back every character received (a way to insure error-free transmission), which is called *echoing*, we select as the operating method `Full-Duplex`.

Now we connect the modem to the serial connector on the ST. To start with telecommunications, we must run a suitable program which handles the transmission. The simplest method is to call the VT-52 terminal emulator of the ST. Every character typed in is automatically output on the serial interface and all received characters are displayed on the monitor. We can now call a bulletin board, and through the keyboard and screen communicate with the other party.

There is a problem we must contend with. During our conversation with the other party, using the terminal emulator, we can't print out or store the information we are receiving. Since many bulletin boards send introductory texts or help menus, much information can be lost if you are not able to read as fast as the lines disappear from the screen.

Therefore we must give up the ease of use of the emulator and write a small program ourselves which supports communication through the modem and at the same time stores the received text. Such a program must constantly interrogate the RS-232 interface and the keyboard to determine if a character is present. In this way, two-way (full duplex) operation can be maintained in which transmission and receipt are executed almost simultaneously.

We have included a machine language program that checks whether a character is from the keyboard or the RS-232 port. A flag checks the direction of the data.

```
10    rem ** Terminal Program 7.6.3 **
15    fullw 2: clearw 2 : gotoxy 0,0
20    i = 0
30    dim a%(5000)       : rem Reserve Storage
100   gosub 2000         : rem m/l routine
170   goto 100           : loop
180   for j=1 to i             : rem Yes, all Output
185 if a%(j) = 13 then print:
190   print chr$(a%(j))
200   next j
300 end
```

```
1000 dim co%(34)          :rem m/l loader
1010 for y = 0 to 34: read co%(y): next
1490 data &h2a48,&h3f3c,&h000b,&h4e41,&h548f
1500 data &h4a40,&h661e,&h3f3c,&h0012,&h4e41
1510 data &h548f,&h4a40,&h67e8,&h3f3c,&h0003
1520 data &h4e41,&h548f,&h2b40,&h0040,&h426d
1530 data &h0044,&h4e75,&h3f3c,&h0007,&h4e41
1540 data &h548f,&h2b40,&h0040,&h3b7c,&hffff
1550 data &h0044,&h4e75,&h0000,&h0000,&h0000
2000 d = varptr(co%(0)): call ad
2010 if co%(34) = -1 then gosub keyb else gosub v.24
2015 i = i + 1: a%(i) = x : rem store
2020 return
3000 keyb:
3010 sc = 0:x=co%(33): if x = 0 the sc =1:x=co%(32)
3014 if x = asc("*") then 180: rem end
3015 if x = 13 then print
3016 print chr$(x);
3020 out 1,x
3030 return
4000 v.24:
4005 x = co%(33)
4010 if x = 13 then print
4020 print chr$(x);
4030 return
```

In lines 10 to 40 some preparations are made. Lines 100-170 are the main loop. Lines 180-300 print out the recieved data. Lines 1000-1550 are the loader for the machine language. Lines 2000-4020 reads and displays data from the keyboard or RS-232 port using the machine language routine.

If only an asterisk (*) is input, the program is terminated and the text is redisplayed. At this point a PRINT or an LPRINT can be used to send the output to the printer and produce hardcopy of the session.

7.7 Character editors

A *Font* is the common term for the character set. The font set of the ST contains an enormous variety of characters and symbols. The Greek letters are available, which is helpful for physicists. Also mathematical symbols such as √ , ∫, or also ↔ are available. You can provide scientific text on the screen with the exact formulas.

For some applications you can define your own symbols. By doing this, you can program games for example, by using a sequence of `PRINT CHR$()` statements which increases the speed of the graphics display and at the same time simplifies the programming.

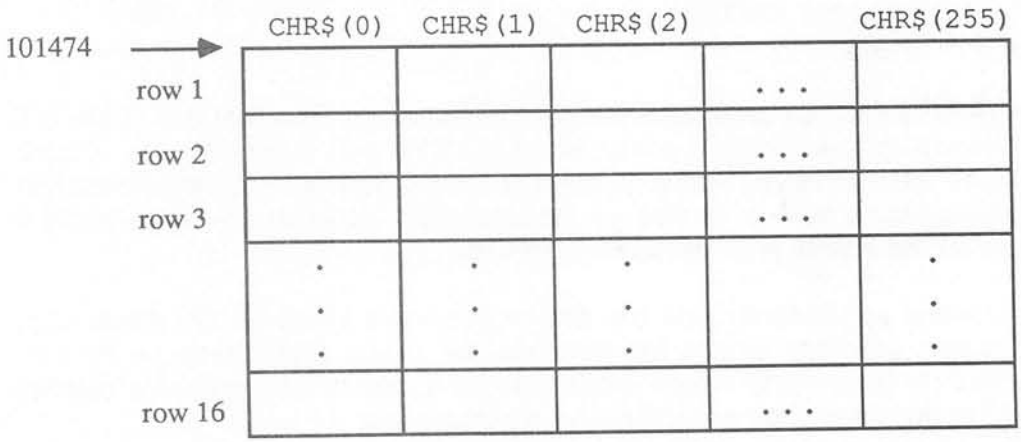
BASIC does not offer any way to define your own symbols. Instead you have to use `PEEK` and `POKE` to change the font.

The operating system has 3 built in fonts. They are identical in design, but differ in resolution. For the display on the monochrome monitor there is a 8*16 character set, i.e. every character is 8 dots wide and 16 dots high. For the color monitor it is only 8*8 since the vertical resolution is only half as great. And finally there is the 6*6 font for identifying icons or the files in the table of contents.

To change one of these fonts, you must know where they are stored.

In the version of the operating system that is loaded from diskette, the 8*16 font is located at 101474. In the version of the operating system that is stored in ROM, the 8*16 font is located at 1659294.

The fonts are arranged in memory in rows. The first row of each of the 256 characters is stored in the first 256 bytes of font memory. The second row of each of the characters is stored in the next 256 bytes of font memory.



Organization of 8 x 16 font in memory

The most significant bit of the character determines the left upper point of the character. In the standard character set, the topmost row is always zero, i.e. white. This prevents two characters from overlapping.

Now to a program which changes the character output by `PRINT CHR$(1)`. This character is the up-arrow character. The symbol can also be obtained by typing <Control> A.

The program again uses the binary decimal conversion which we already used. We'll once again use the image of a little man. Of course you can change this character as you like.

Important: This program only works when TOS is loaded into RAM from disk, it will not work with TOS in ROM! The programs in this section only work on the monochrome monitor.

```
10    rem *** Change Font Character 7.7 ***
20    def seg=1          : rem Set Byte brightness
80    for i=0 to 12 : read x$: rem Fill Shading
82    x=0 : for j=1 to 8             : rem Change
84    x= x-(mid$(x$,j,1)<>)* 2^(8-j)
86    next j
90    poke 105436+i*256,x:rem TOS on disk
100   next i
110   print chr$(1)                  : rem test output
```

```
120   end
130   rem +++ Test Data +++
150   data "   ***   "
160   data "* *****  "
170   data "*  ***   "
180   data "**  *    "
190   data " ******* "
200   data "  **** * "
210   data "  **** * "
220   data "  ****   "
230   data "  *  *   "
240   data "  *  *   "
250   data "  *  *   "
260   data "  *  *   "
270   data "  ** **  "
```

If you want to change other characters, you only have to change the value 101474 in line 90. To create a new A, i.e. `CHR$(65)` just add a 64 to address 101474. You can create your own character set which you can then store with `BSAVE"MYFONT.FNT",104536,4096` on the disk and load later with `BLOAD"MYFONT.FNT",104536`.

This filename is not mandatory. You can store many fonts with files saved in this manner and so prepare fonts for every application. Games can be written or text prepared in the Cyrillic alphabet. The applications of these fonts have no limit.

Some advice before you create your own characters, first store the original font. If you do not need your characters any more, you can load the the original font so that you can read the disk directory and the menu choices.

There is a more general way to change a character set. This method involves copying the original font into RAM and changing the operating system pointer to the font. This pointer is utilized by the TOS when accessing font data. This method has two advantages:

1) When the original character set lies in ROM (latest versions of the ST), it cannot be altered. Once it is copied to RAM, however, the copy can be edited, saved and reloaded for later use.

2) The pointer which is used in the following example works with the OUT function only.

Thus, you can have two different character sets, say one for PRINT and one for OUT 2,x. Both fonts can also be combined. The program below will work with TOS in ROM or RAM, simply change line 20. Here is an application for the above mentioned method:

```
1    rem monochrome monitor only
5    rem *** Character Editor 7.7.A ***
10   defdbl a,b,c : rem pointer preparation
20   a=16595294: rem address of the original
     font/TOS in ROM change to a=104536 for
     TOS on disk
30   c=10532  : rem address of the pointer
40   x=a      : rem copy font-pointer
50   dim a%(2050)  : rem reserve position
60   for i=0 to 2050
70   a%(i)=peek(x+i*2) : rem copy font data
80   next i
100  c$="A"   : rem to another screen
110  n=&H66   : rem bits of the top line
120  b=varptr(a%(0)): rem address of the new
     font
130  poke c,b      : rem change pointer
140  m=&HFF        : rem screen preparation
150  o=asc(c$)
160  if (o mod 2)=1 then m=m*256 else
     n=n*256
170  a%(o/2)=a%(o/2) and m or n
180  gosub 300 : rem modify test text
190  poke c,a  : rem original font placement
200  gosub 300 : rem original test text
210  end
300  restore
310  for i=1 to 9:rem output test text 'ABC'
320  read x
330  out 2,x  : next i
335  for i = 1 to 1000: next
340  return
350  data 27,72,27,66,27,66,65,66,67
```

The first section of this program prepares the necessary pointers, then copies the original font into the integer array a%. Note that every entry of the character set requires two bytes in this array. For this reason, a mask must be created when accessing a character; this mask will hide the extra byte so that the byte you want changed will not be influenced. This mask is set up in the second program section. This mask sets apart the second byte, and overwrites it with the value n. For example, if the value $66 is used, the A is altered.

The third part of this example displays ABC as the altered text. After using the new font, the sample text will be output again, and will appear as normal. This example shows just how easy it is to create a new character set. Installing the new font data for applications is somewhat more difficult. There is help, though. The program below is a simple character editor which, although slow (in BASIC), will allow you to edit fonts using the mouse.

Starting the program displays a pattern of 8 x 16 asterisks onscreen. The actual character appears next to this matrix, and shows the actual size of the character. Thus you have direct control over the appearance of the new character. An asterisk in this matrix represents one set (black) point, or pixel, in the real character. A character is built within a 128-pixel matrix (8 pixel columns x 16 pixel rows).

The original character can be displayed within this pattern, although they will appear unaltered with any PRINT statement. You will see "<" and ">" symbols beneath the original character. Click this symbol to display the previous or next character.

Character editing is fairly simple here. Clicking an asterisk on in the 8 x 16 matrix also sets a corresponding pixel in the original character.

```
10   rem  Character editor 7.7.B Mono only
20   dim a%(2050)
30   defdbl a,b,c
40   a=16595294: c=10532
50   d = a
60   for i = 1 to 2050
70   a%(i) = peek(d+i*2)   : 'font copier
80   next i
90   b = varptr(a%(0))   : 'loc. of new data
100  fullw 2 : clearw 2
```

```
110  poke c,b        : 'install new font
120  ch = 65         : 'start with A
130  poke contrl, 123 : vdisys 0:' mouse off
140  gotoxy 12,2
150  ?"character : ",chr$(ch):'original char
160  gotoxy 11,4 : ?" <   >"
170  if (ch mod 2)=0 then o = 8 else o=0
180  gotoxy 0,1 : ? "__________"
190  for i = 1 to 15       : 'draw char
200  gotoxy 0,i+1 : ? "|";
210  x = a%(ch/2+128*i)
220  for j = 0 to 7
230  if (x and 2^(7-j+o)) then ?"*"; else
     ?"-";
240  next j : ?"|"
250  next i
260  ?"----------"
270  out 2,27 : out 2,asc("Y")
275  out 2,36 : out 2,45 : ' cursor position
280  for i =1 to 5 : out 2,ch : rem 5 new
     characters
285  next i
290  poke contrl ,122 : vdisys 0 : 'mouse on
300  mx = int(peek(9952)/8)-1
310  my = int(peek((9954))/16)-4: ' mouse
     position
320  if (peek(3581) and 1)=1 then 420 :
     rem right button
330  if (peek(3581) and 2)=0 then 290 :
     rem left button
340  if my =3 and mx = 24 then ch = ch - 1 :
     goto 130
350  if my = 3 and mx = 27 then ch = ch + 1
     : goto 130
355  if my < 0 then 300
356  if my >15 then 300
360  if mx>8 then 300
370  x = a%(ch/2+128*my)
380  m =2 ^(7-mx+o)      : 'selected bit
390  if (x and m)=0 then x = x or m else
     x=x and (m xor &HFFFF)
400  a%(ch/2+128*my)=x    : ' renew
```

```
410  goto 130
420  gotoxy 0,18
430  input "File name :",f$ : 'font save
440  bsave f$,b,4100
```

Lines 420-440 save the new character set to disk. You click the right mouse button to save the character set. You can re-load this character set and re-activate it using the pointer at 10532. Now nothing stands in your way to keep you from creating exotic fonts and character sets for your programs!

7.8 The keyboard buffer.

The keyboard buffer in the Atari St can be accessed with the PEEK function. The keyboard buffer does not return an ASCII value, but instead it returns a scan code.The following program outputs the scan code

```
10   rem ** Keyboard buffer 7.8**
20   p=3510     : rem Pointer to character
30   x=peek(p)       : rem Temporary storage
40   gotoxy 10,1 : print y:y=y+1
     : rem example
50   if peek(p)=x then 30: rem New character
60   x=peek(p)       : rem Yes, New pointer
70   t=peek(3086+x) and 255 : rem Get
     character
80   gotoxy 1,1: print t : rem scan code
90   goto 40             : rem Infinite loop
```

Every character typed at the keyboard is automatically stored in a buffer by the operating system. This buffer begins in location 3086. The information about which character is currently in the buffer is stored in location 3510. This pointer is read by line 30 and is stored in variable x. If another character is entered, this pointer is increased by 2. This change is noted in line 50. The new pointer is read out and added to the start of the buffer. The new character is obtained with PEEK(3086+X). The scan code is output and the program continues. You can clearly see in this example that the program does not wait for a key to be pressed, but runs continuously.

Appendices

Appendix A

Glossary

Addresses

The individually addressable, sequentially numbered storage locations of the working storage area. These location numbers permit access (read/write) to the contents of the storage location and represent its address.

Application

An program that is directly executable.

ASCII (American Standard Code of Information Interchange)

A standard code set that assigns each alphanumeric character a binary number. This is the most commonly used code set, in either the 7- or 8-bit form with 128 or 256 characters.

Assembler

A program that translates machine language programs written in mnemonic code into the object code of the microprocessor.

Baud

Baud is a measure of speed for serial data transmission on an RS-232 interface. 300 baud means a transmission speed of 300 bits per second. Speeds of 300 to 9600 baud can be selected with the control panel of the Atari ST.

Bit (Binary Digit)

Computers are equipped internally to work only with binary numbers, i.e. numbers composed of 0 and 1. Binary digits (bits) are the smallest information units in computer technology.

Buffer Storage

A buffer represents a storage area in which data is stored temporarily. Processes of different speeds can be combined with buffers, where the faster process stores its results in the buffer and the other process can later read them out at its own speed (for example a printer buffer).

Bus

A bus is a system of lines between the individual components of a computer, for example between CPU and working storage, or between several units such as computer and printer, such as the peripheral bus. There are two different kinds of buses unidirectional (data transmission in one direction) and bidirectional (data transmission in two directions).

Byte

A byte is an 8 digit binary number. It represents the smallest addressable data unit, even if the CPU and storage is organized into two byte words or two longwords as in the Atari ST.

C Programming Language

C is a higher level language that retains many of the characteristics of machine language. It is very fast, close to the hardware and fairly simple to learn. The structure of the language is based on ALGOL and PASCAL. Programs written in C are easily transferred to other computers if a C compiler exists for the target machine. Using this language, it is now possible that operating systems (for example UNIX or GEM) can be implemented in different CPUs without major rewriting.

Cartridge

Cartridges contain up to 128K of read-only storage which are inserted into the slot on the left side of the Atari ST. They contain application programs or extensions to the operating system.

Centronics Interface

This interface was introduced by the Centronics company for connecting printers and has established itself as a standard. It is a unidirectional 8 bit wide parallel interface with a handshake line. In the Atari ST this interface is bidirectional.

Clicking

Selection of a GEM symbol (Icon) by touching it with the mouse pointer and pressing the mouse key.

Control Character

For execution of special functions with printers or terminals the non-printing ASCII codes from 0 to 31 or control sequences (the <control> key following by any other key) are used.

Control Panel

Dialog box available in the desktop in which different parameters can be set such as color, time of day, key noise, etc.

CPU (Central Processing Unit)

In microcomputers the microprocessor is called the CPU. The Atari ST uses the MC 68000 from Motorola.

Cursor

The cursor is a differently formed symbol (arrow, bee, crosshairs or block) which can be moved on the screen with the mouse (mouse cursor) or keyboard (text cursor) and marks the next input position.

Debugger

A utility program for error detection and correction in assembled and compiled machine language programs. Break points can be set and the memory contents read and changed. (Symbolic Interactive Debugger or SID)

Desktop

The main display level under GEM which contains menu lists, disk icons and the waste basket.

Dialog box

An interactive window that provides information to the user and waits for a response.

Disassembler

A utility program capable of translating a machine language program back to mnemonic code. Error correction and changes are facilitated.

DMA

DMA means Direct Memory Access and designates the capability of peripheral units to write or read data in memory without the participation of the CPU.

Duplex

A data transmission method in which it is possible to transport data in two directions at the same time (Full Duplex) i.e. it only takes a short time to change direction.

Emulation

The process where one computer is simulated by another computer with the help of software and/or additional hardware. For example, on the Atari ST in terminal mode, a VT52 terminal is emulated.

File

A file is a data group in memory, on a disk or a hard disk. The access to this data occurs under a specific name.

Folder

In the table of contents of the disk (directory), subdirectories are called folders. These can contain files which are not shown in the higher level table of contents. Access to these files is only possible after opening the folder.

Formatting

Before data can be written on a new disk, it must be formatted. i.e. tracks and sectors must be set up.

GDOS (Graphic Device Operating System)

Contains the device independent graphic functions of the GEM-VDI.

GIOS (Graphic Input/Output System)

This part of GEM-VDI contains the device dependent code.

Hardcopy

Output of the actual screen contents on the printer.

Hard Disk

A storage device which operates on the same magnetic principle as a floppy disk. The difference lies in the storage capacity of the hard disk (5 to 500 megabytes) and the transmission speed. This is achieved by the fixed installation of the "disk" which spins at a higher speed than the floppy.

Hexadecimal

The representation of numbers in the base 16 system. This is the most common number system besides the binary and octal systems in computer science.

I/O (Input/Output)

These concepts describe the data read from peripherals (mass storage, keyboard, mouse, etc.) into the computer or are sent from the computer to the peripherals (display, printer, plotter, etc.)

Interface

The electronic connection circuitry between computer and peripheral equipment (for example Centronics or MIDI). It can also be a program which standardizes the connection between differing, independent programs or the user and programs (for example GEM between user and TOS).

Interrupt

An interruption of the executing program and branching to a machine language routine. After its processing, the interrupted program is continued at the same location where it was halted (just as in a subroutine). This interrupt is triggered by a hardware event at a certain port of the CPU or through software by means of a program (for example with a `TRAP` command).

K (Kilobyte)

Kilobyte really means 1,000 bytes, but in the computer field we calculate in binary powers; a Kilobyte is therefore 2^{10} = 1,024 bytes.

Library

In data processing, a collection of subroutines, functions or utility programs which can be included into a program.

Linker

A utility program which links together compiled programs, machine code routines and parts of the library into a program capable of execution.

Menu

A menu in a program provides several choices from which one can be selected with keyboard input or the mouse pointer.

Memory Address (Address)

The locations of the computer memory are numbered consecutively so that only one of the memory locations can be selected with the number provided.

MIDI Interface

A standardized, serial interface for controlling musical instruments from the computer. Several instruments can be controlled simultaneously thru the ST's MIDI port.

Mnemonic Code

The actual machine code consists exclusively of ones and zeros. Since commands consisting of only 0's and 1's would be difficult to read, easily remembered alphabetic combinations were introduced. For example, the sequence of ones and zeros which adds two numbers was changed to the mnemonic code ADD. Programs written in mnemonic code have to be translated by assemblers in order to create executable programs.

Object code

Represents program code (consisting of bit patterns or hexadecimal numbers) which is directly executable on the CPU. This code is obtained through compilation of a higher level language program or assembly of mnemonic code.

Output

Transmission of data to a peripheral unit.

Parallel Interface

See Centronics Interface

Parameter

Variables or constants which are passed to commands, functions, or subroutines for processing. Several parameters may be needed for one function, for example LINEF *A, B, C, D*.

PEEK

The BASIC command PEEK reads the contents of a specified mermory location and transfers it to the calling program. This command can process bytes, words or longwords.

Peripheral

The external units of a computer system such as printer, diskette drive or display are called peripherals.

Pixel

The smallest addressable graphic element on the screen (picture point) or printer (matrix point).

POKE

This BASIC command is the opposite of the PEEK command. Memory locations can be directly changed with this instruction. This command also processes bytes, words and longwords. An erroneous POKE command can crash the system if an access to the operating system results.

RAM (Random Access Memory)

This is memory that can be read from and written to.

Register

Describes the internal memory areas in a processor in which data is not only stored, but combined. The MC 68000 has a total of 16 registers (each with 32 bits) for use by the programmer.

RGB (Red - Green - Blue)

A video signal in which the three color signals are sent individually to the television or monitor where they are additively mixed. The Atari ST can represent each of these basic colors in 8 intensity steps to produce 8*8*8 = 512 color mixtures.

ROM (Read Only Memory)

ROM designates memory which, in contrast to RAM, can only be read. The programming of these chips occurs during production. They are often used to store the operating system in a computer so that it will be available immediately after powering up the unit.

RS 232

This standardized interface works serially. The data transmission can be performed in two directions. The signal strength used is + and -12 volts.

Scrolling

Shifting a window's contents in one of the four basic directions. This shifting is performed by the operating system, when the user clicks the vertical or horizontal shifters of a window with the mouse.

TOS (Tramiel Operating System)

The operating system used in the Atari ST is an enhancement of CP/M 68K with some additional functions. TOS is not compatible with other computers .

VDI (Virtual Device Interface)

A portion of GEM which is responsible for graphic output to any desired peripheral unit.

VT52 Terminal

This choice of the desk menu causes the ST to emulate a VT52 terminal, which can be directly connected through a serial interface to a modem or to another computer.

Word

On the Atari ST a word is a data unit which consists of 2 bytes, i.e. 16 bits, and can have a value from 0 to 65,535.

Appendix B

Important PEEKS and POKEs for Disk and ROM versions of TOS

The following is a list of corresponding PEEK and POKE addresses. The list can be used to adapt the programs in this book to the various ST operating systems.

The first column is for the 197K operating system with the date November 1985, the most common version loaded from disk.

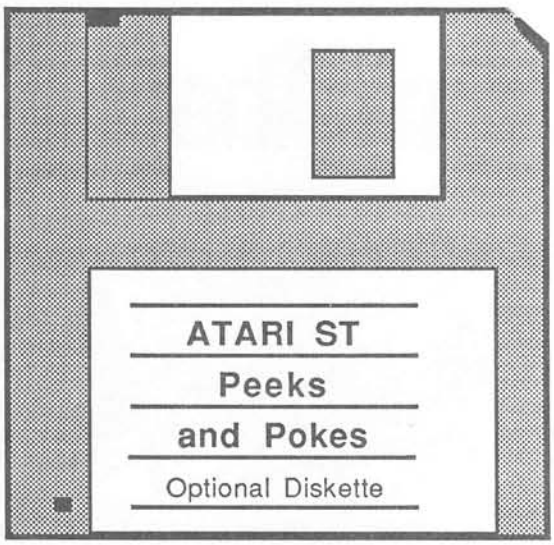
The second column corresponds to the address which applies to the orginal version of TOS, 207K and is also loaded from disk.

The third column is to be used if the TOS is built into the computer. This applies to the April 1986 version. These values are used in this book.

197K Address	207K Address	ROM Address	Function
3086	2256	3086	keyboard buffer
3510	2552	3510	pointer to keyboard buffer
3581	2623	3581	mouse/joystick button status
3582	2624	3582	joystick position
3584	2626	3584	clock buffer(6 bytes)
3591	2633	3592	joystick status
3652	2694	3652	sound sequence data pointer
9952	8994	9952	mouse position horizontal
9954	8996	9954	mouse pos vertical
10530	9572	10530	cursor blink delay
10531	9573	10531	cursor blink counter
10532	9574	10532	pointer to font data
36612	32090	16527388	sound data bell
36642	32120	16527418	sound data click
104536	101474	16595294	8x16 font start

Index

Optional Diskette



For your convenience, the program listings contained in this book are available on an SF354 formatted floppy disk. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

Abacus Software
P.O. Box 7219
Grand Rapids, MI 49510

Or for fast service, call **1- 616 / 241-5510**.

Top shelf books from Abacus

PRESENTING THE ST
Gives you an in-depth look at this sensational new computer. Discusses the architecture of the ST, working with GEM, the mouse, operating system, all the various interfaces, the 68000 chip and its instructions, LOGO. 200pp $16.95

ST Beginner's Guide
Written for the firsthand ST user. Get a basic understanding of your ST. Explore LOGO and BASIC from the ground up. Simple explanations of the hardware and internal workings of the ST. Illustrations, diagrams. Glossary. Index. 200pp $14.95

ST INTERNALS
Essential guide to the inside information of the ST. Detailed descriptions of sound and graphics chips, internal hardware, I/O ports, using GEM. Commented BIOS listing. An indispensible reference for your ST library. 450pp $19.95

GEM Programmer's Ref.
For serious programmers needing detailed information on GEM. Presented in an easy-to-understand format. All examples are in C and assembly language. Covers VDI and AES functions. No serious programer should be without. 410pp $19.95

MACHINE LANGUAGE
Program in the fastest language for your ATARI ST. Learn 68000 assembly language, its numbering system, use of registers, structure & important details of instruction set, and use of internal system routines. Geared for the ST. 280pp $19.95

ST TRICKS & TIPS
Fantastic collection of programs and info for the ST. Complete programs include: super-fast RAM disk; time-saving printer spooler; color print hardcopy; plotter output hardcopy; creating accessories. Money saving tricks and tips. 260pp $19.95

ST GRAPHICS & SOUND
Detailed guide to graphics and sound on the ST. 2D & 3D function plotters, Moiré patterns, graphic memory and various resolutions, fractals, recursion, waveform generation. Examples written in C, LOGO, BASIC and Modula2. 250pp $19.95

ST LOGO GUIDE
Take control of your ST by learning ST LOGO—the easy to use, powerful language. Topics include: file handling, recursion-Hilbert & Sierpinski curves, 2D and 3D function plots, data structure, error handling. Helpful guide for ST LOGO users. $19.95

ST PEEKS & POKES
Enhance your programs with the examples found within this book. Explores using different languages BASIC, C, LOGO and machine language, using various interfaces, memory usage, reading and saving from and to disk, more. 280pp $16.95

BASIC Training Guide
Thorough guide for learning ST BASIC programming. Detailed programming fundamentals, commands descriptions, ST graphics & sound, using GEM in BASIC, file management, disk operation. Tutorial problems give hands on experience. 300pp $16.95

BASIC to C
Move up from BASIC to C. If you're already a BASIC programmer, you can learn C all that much faster. Parallel examples demostrate the programming techniques and constructs in both languages. Variables, pointers, arrays, data structure. 250pp $19.95

3D GRAPHICS
FANTASTIC! Rotate, zoom, and shade 3D objects. All programs written in machine language for high speed. Learn the mathematics behind 3D graphics. Hidden line removal, shading. With 3D pattern maker and animator. $24.95

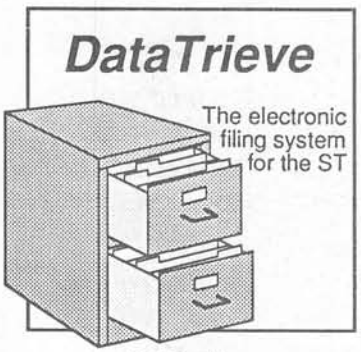
Abacus Software

P.O. Box 7219 Dept. A9 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Optional diskettes are available for all book titles at **$14.95**

Call **now** for the name of your nearest dealer. Or order directly from ABACUS with your MasterCard, VISA, or Amex card. Add $4.00 per order for postage and handling. Foreign add $10.00 per book. **Other software and books coming soon.** Call or write for your **free** catalog. Dealer inquiries welcome–over 1400 dealers nationwide.

AA Rated Software Atari and Abacus

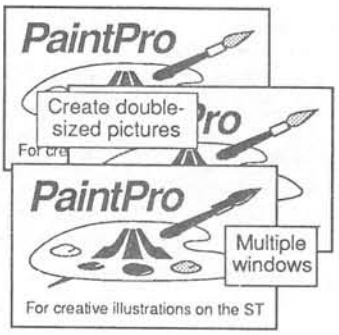


ST DataTrieve

A simple-to-use and versatile database manager. Features help screens; lightning-fast operation; tailorable display using multiple fonts; user-definable edit masks; capacity up to 64,000 records. Supports multiple files. RAM-disk support for 1040ST. Complete search, sort and file subsetting. Interfaces to TextPro. Easy printer control. **$49.95**

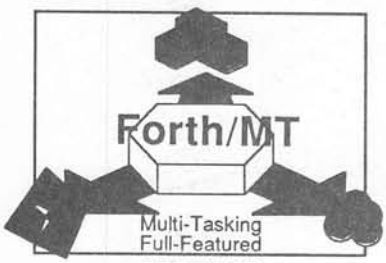
ST TextPro

Wordprocessor with professional features and easy-to-use! Full-screen editing with mouse or keyboard shortcuts. High speed input, scrolling and editing; sideways printing; multi-column output; flexible printer installation; automatic index and table of contents; up to 180 chars/line; 30 definable function keys; metafile output; much more. **$49.95**



ST PaintPro

A GEM™ among ST drawing programs. Very friendly, but very powerful design and painting program. A *must* for everyone's artistic or graphics needs. Use up to **three** windows. You can even cut & paste between windows. Free-form sketching; lines, circles, ellipses, boxes, text, fill, copy, move, zoom, spray, paint, erase, undo, help. Double-sized picture format. **$49.95**

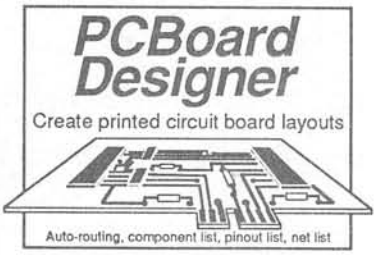


ST Forth/MT

Powerful, multi-tasking Forth for the ST. A complete, 32-bit implementation based on Forth-83 standard. Development aids: full screen editor, monitor, macro assembler. 1500+ word library. TOS/LINEA commands. Floating point and complex arithmetic. **$49.95**

ST AssemPro

Professional developer's package includes editor, two-pass interactive assembler with error locator, online help including instruction address mode and GEM parameter information, monitor-debugger, disassembler and 68020 simulator, more. **$59.95**



PCBoard Designer

Interactive, computer aided design package that automates layout of printed circuit boards. **Auto-routing**, 45° or 90° traces; two-sided boards; pin-to-pin, pin-to-BUS or BUS-to-BUS. Rubber-banding of components during placement. Outputs pinout, component and net list. **$395.00**

Call **now** for the name of the dealer nearest you. Or order directly using your MC, Visa or Amex card. Add $4.00 per order for shipping. Foreign orders add $10.00 per item. Call (616) 241-5510 or write for your **free** catalog. 30-day money back software guarantee. Dealers inquires welcome–over 1400 dealers nationwide.

Abacus

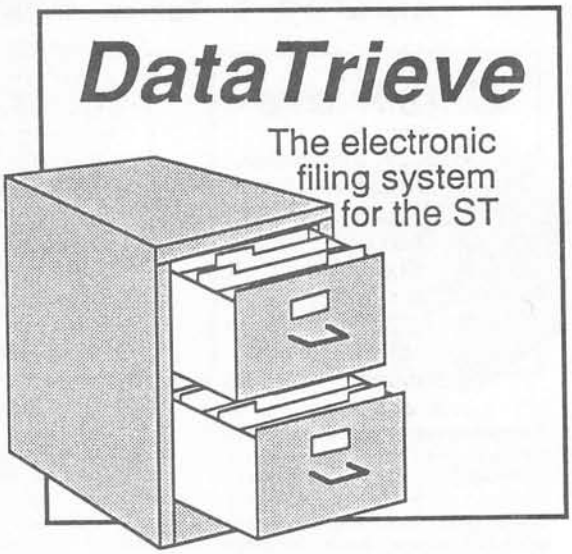
P.O. Box 7219 Dept.NB Grand Rapids, MI 49510
Phone 616/241-5510 • Telex 709-101 • Fax 616/241-5021

ST DataTrieve

for the Atari ST

(formerly ST FilePro)

ST DataTrieve is a simple-to-use, versatile database program. DataTrieve's drop-down menus let you quickly define a file and enter your information through screen templates. DataTrieve allows you to store data in different type styles, create subsets of a file, change file definition and format, and do fast searches and sorts. A RAM disk is supported on the 1040ST, as well as multiple files. DataTrieve also features hardcopy to most dot-matrix printers (Epson and compatible). If your printer is not one of those listed, the printer driver is easily adapted by the user to his printer model. DataTrieve even supports text effects, and contains an integral list editor, to print out either the data file itself or a list, such as a mailing list.



OTHER FEATURES OF ST DataTrieve:

* four files can be open simultaneously
* maximum file size of 2,000,000 characters
* maximum data set size of 64,000 characters
* maximum of 64,000 data sets
* unlimited number of data fields
* mass-memory-oriented file organization
* up to 20 index fields per file
* unlimited number of search criteria
* data exchange with other programs possible
* maximum screen mask size of 5000 x 5000 pixels
* text editor-like mask input

Suggested Retail Price:

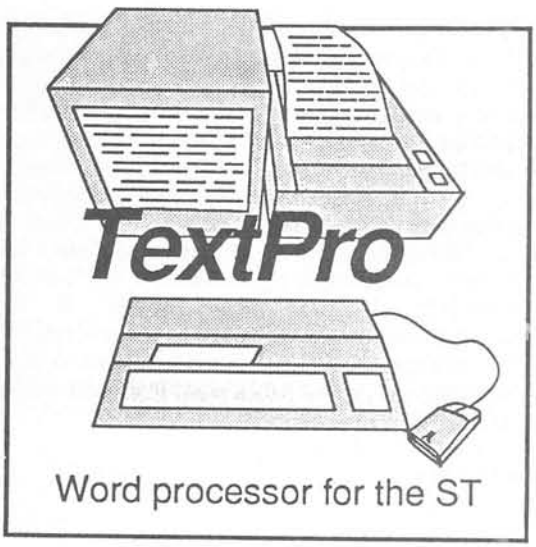
$49.95

ABACUS SOFTWARE
P. O. Box 7219
Grand Rapids, MI 49510
Phone: (616) 241-5510

ST TextPro

for the Atari ST

ST TextPro is the professional wordprocessing package designed for the ST by professional writers. ST TextPro combines great features with flexibility, speed and easy operation—but at a very reasonable price! ST Textpro offers full-screen editing with mouse or keyboard shortcuts, as well as high-speed input, scrolling and editing. The authors designed TextPro for professionals and two-digit typists alike. ST TextPro includes a number of practical formatting commands, fast and easy cursor manipulation and text enhancement. Features include a C-source mode, which allows you to write C program code in TextPro format; mail merging; and much more.

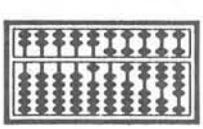


OTHER FEATURES OF ST TEXTPRO:

* up to 180 characters per line with horizontal scrolling
* up to 30 user-assignable function keys, with up to 160 characters per key
* any number of tabulators
* automatic hyphenation
* up to 5-column output (printed sideways)
* DIN A4 vertical printout for Epson FX and compatibles
* flexible printer driver
* RS-232 file transfer possible (computer-computer)
* detailed manual
* TextPro files can be set for layout with ST TextDesigner (available separately)

Suggested Retail Price:

$49.95

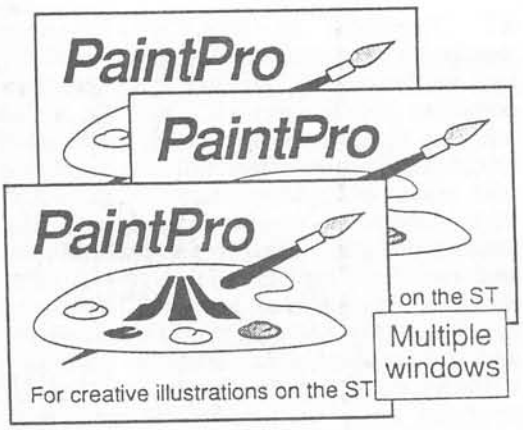


ABACUS SOFTWARE
P. O. Box 7219
Grand Rapids, MI 49510
Phone: (616) 241-5510

ST PaintPro

for the Atari ST

ST PaintPro is a very friendly and very powerful package for drawing and design. Based on GEM™, PaintPro supports up to three active windows and has a complete toolkit of functions, including drawing, lines, circles, rectangles, fill, spray, and others. Text can be typed in in one of four directions (forward, up, down, backward) and in one of six GEM fonts and eight sizes. You can even load pictures from other formats, such as ST LOGO, DEGAS and DOODLE for enhancement, using PaintPro's double-sized picture format. Hardcopy can be sent to most popular dot-matrix printers. Works with either monochrome or color ST systems.

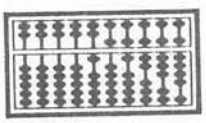


OTHER FEATURES OF ST PAINTPRO:

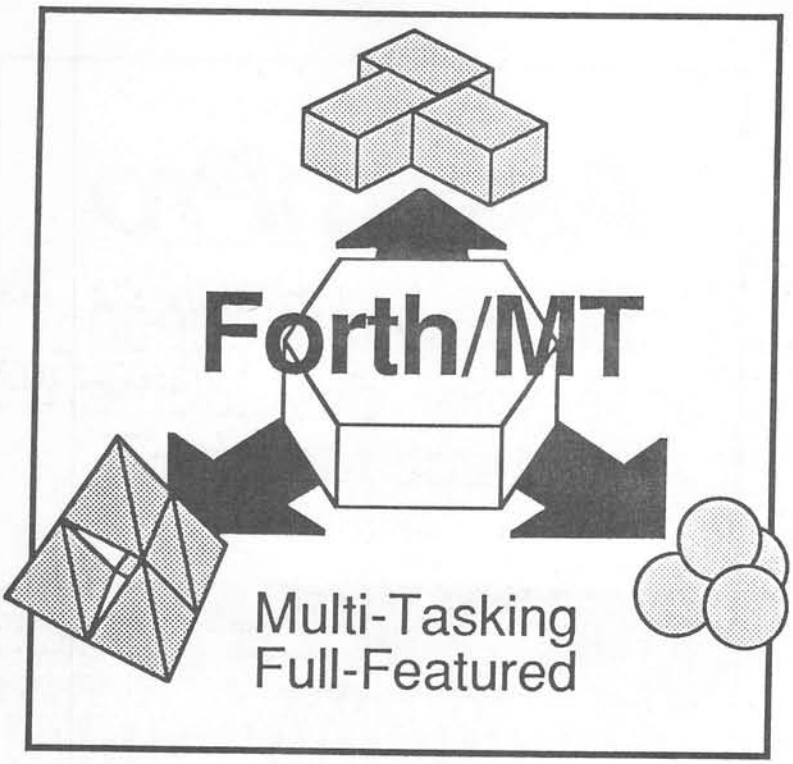
* four drawing modes (replace, transparent, inverse and XOR)
* four line thicknesses
* all Atari ST patterns
* maximum of three windows (depending on available memory)
* resolution of up to 640 x 400 or 640 x 800 (DIN A4) pixels (monochrome version only)
* up to six GEM type fonts, in 8-, 9-, 10-, 14-, 16-, 18-, 24- and 36-point sizes; text can be printed in four directions
* blocks can be cut and pasted; mirrored horizontally and vertically; marked, saved in LOGO format, and recalled in LOGO
* most dot-matrix printers can be easily adapted by the user
* accepts LOGO, DEGAS and DOODLE graphic

Suggested Retail Price:

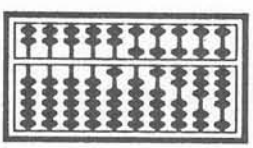
$49.95



ABACUS SOFTWARE
P. O. Box 7219
Grand Rapids, MI 49510
Phone: (616) 241-5510



Forth/MT, the multi-tasking, full-featured Forth language for the ST, is for serious programmers. **Forth/MT** is a complete, 32-bit implementation based on Forth '83 standard. Includes many development aids: full screen editor, monitor, macro assembler, over 1500 word library. Includes TOS, LINEA, floating-point and complex arithmetic commands. $49.95



ABACUS SOFTWARE
P. O. Box 7219
Grand Rapids, MI 49510
Phone: (616) 241-5510

PEEKS & POKES

Keys to revealing the secrets hidden within your Atari® ST™

Unlock the hidden secrets within your ST with **PEEKS & POKES.** The PEEK and POKE commands are a bridge between you and your ST's operating system through ST BASIC. This book gives you a closer look at the many functions of your ST. Part of the collection of "quick hitters" and information packed inside include:

- Customizing the desktop
- Changing the mouse shape
- Changing the character sets
- Using the mouse as a paintbrush
- Reading the keyboard or joystick
- Important PEEKs & POKEs
- Making your own fill patterns
- Direct disk access
- Number systems
- System variables
- Internal memory
- Interpreter and compiler
- Pointers and the stack
- ST communications
- Setting the RS-232 interface

About the author:
Stefan Dittrich is a computer science major and ST expert. He was one of the first to own an ST developer's package, and uses it to develop new software and hardware hints for Data Becker.

ISBN 0-916439-56-9

Part of the continuing series of informative books from

you can count on Abacus

A Data Becker Book

ATARI
ST

ATARI ST Peeks & Pokes

Dittrich

ABACUS