THE CONCISE ATARI ST 68000 PROGRAMMER'S REFERENCE GUIDE

Katherine Peel

About the Atari ST Series
The Atari ST is one of the most significant new computers to be launched in recent years. Its performance and technical specifications are outstanding and comparable to machines several times the price. Moreover, it is a machine which appeals to the hobbyist and business user alike and the large amount of software available allows it to be used extensively in the home or office.

This series of books provides Atari ST owners with practical, down-to-earth information about their computers – from the introductory level through to advanced programming techniques and professional business uses.

About this book
The aim of this book is to provide the Atari ST user with a complete reference manual to the machine. It is designed to be used both as a quick reference manual and as a source of detailed technical material. Topics covered include machine code programming, details of GEM and the operating system. Much of the material included in the book has been taken directly from official Atari technical documentation and is unlikely to be found from any other source. The book will be an essential reference manual for every Atari ST owner.

About the author
Katherine Peel was formerly a senior systems analyst in industry, and now works as a freelance author. She has been a major contributor of reviews and technical articles to the "Your Computer" magazine for a number of years, providing in-depth authoritative reviews of the latest hardware.

Other titles in the series
Introducing the Atari ST
Using ST BASIC on the Atari ST
Using GEM on the Atari ST
Using Databases on the Atari ST
Business Applications with the Atari ST
Using Graphics on the Atari ST
Using LOGO on the Atari ST
Practical LOGO on the Atari ST

GLENTOP PUBLISHERS LIMITED

Glentop Publishers Ltd
Standfast House
Bath Place
High Street Barnet
Herts EN5 5XE

ISBN 1-85181-017-X

PUBLISHERS £15.95 PRICE

9 781851 810178

ATARI ST SERIES

Series Editor: Robin Bradbeer
Foreword by Jack Tramiel

THE CONCISE ATARI ST 68000 PROGRAMMER'S REFERENCE GUIDE

Katherine Peel

GLENTOP

Concise Atari 68000 Programmer's Reference Guide

by

K.D. Peel

Concise
ATARI 68000
PROGRAMMER'S
REFERENCE GUIDE
by
K.D. Peel

Glentop Publishers Ltd

Concise ATARI 68000 PROGRAMMER'S REFERENCE GUIDE

by

K.D. Peel

Glentop Publishers Ltd

AUGUST 1986

ISBN 1 85181 017 X

Published by: Glentop Publishers Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 5XE
Tel: 01–441–4130

Originated directly from the publisher's w-p disks by
NWL Editorial Services, Langport, Somerset, TA10 9DG

Contents

Appendices

Preface

This book is intended as a compact reference guide to the Atari ST range of computers, it provides detailed information on the Atari ST hardware, an overview of the operating systems and the operating system calls.

The majority of the book has been prepared in both decimal and hexadecimal notation to make reading and data entry less complicated for the beginner, and those who wish to use the VDI and AES tables from BASIC. I hope the use of decimal will not be too distressful to the purists, but most assemblers will accept either format as an input. The diagramatic presentation of data in memory and of stacks follows the Motorola MC68000 user's manual format of low memory towards the top of the page; presentation of memory maps follows the convention of high memory towards the top of the page. All memory representations are annotated to avoid confusion.

The Atari ST range of computers contain the largest ROM (192K) of all the current home/low cost business computers available. This offers an enormous wealth of data and routines that the user may wish to access; about six times that of most other computers, this information is presented in a condensed group tabular form to provide association. General descriptions of all the facilities available (disk, file, interfaces etc) are provided to present the reader with at least an outline understanding of their operation.

The book covers the programming of an Atari ST 16-bit computer in three parts:

Chapter 1 gives an overview of the Atari ST hardware and expansion ports, also included is a short description of the peripheral interface circuits.

Chapter 2 presents an overview of the operating systems, the management of memory and resources, control of serial I/O, screen functions and file handling.

Chapter 3 provides the operating system calls for both GEM and TOS, the A-line graphic functions and the intelligent keyboard command instructions.

Appendices contain the system variables, configuration registers and a summary of the MC68000 instruction set.

Acknowledgements

The author wishes to thank Atari Corp. (UK) Limited for its assistance in the preparation of this book by providing much of the technical data, which is reproduced with the kind permission of Atari Corp. (UK) Limited.

Disclaimer

Foreword by Jack Tramiel

When we introduced the ST series of computers at Atari, we coined the phrase 'Power without the Price'. This sums up all that had been in our minds when we decided to design a range of powerful but low-cost machines that could be used for all applications ranging from sophisticated games to complex business and scientific uses.

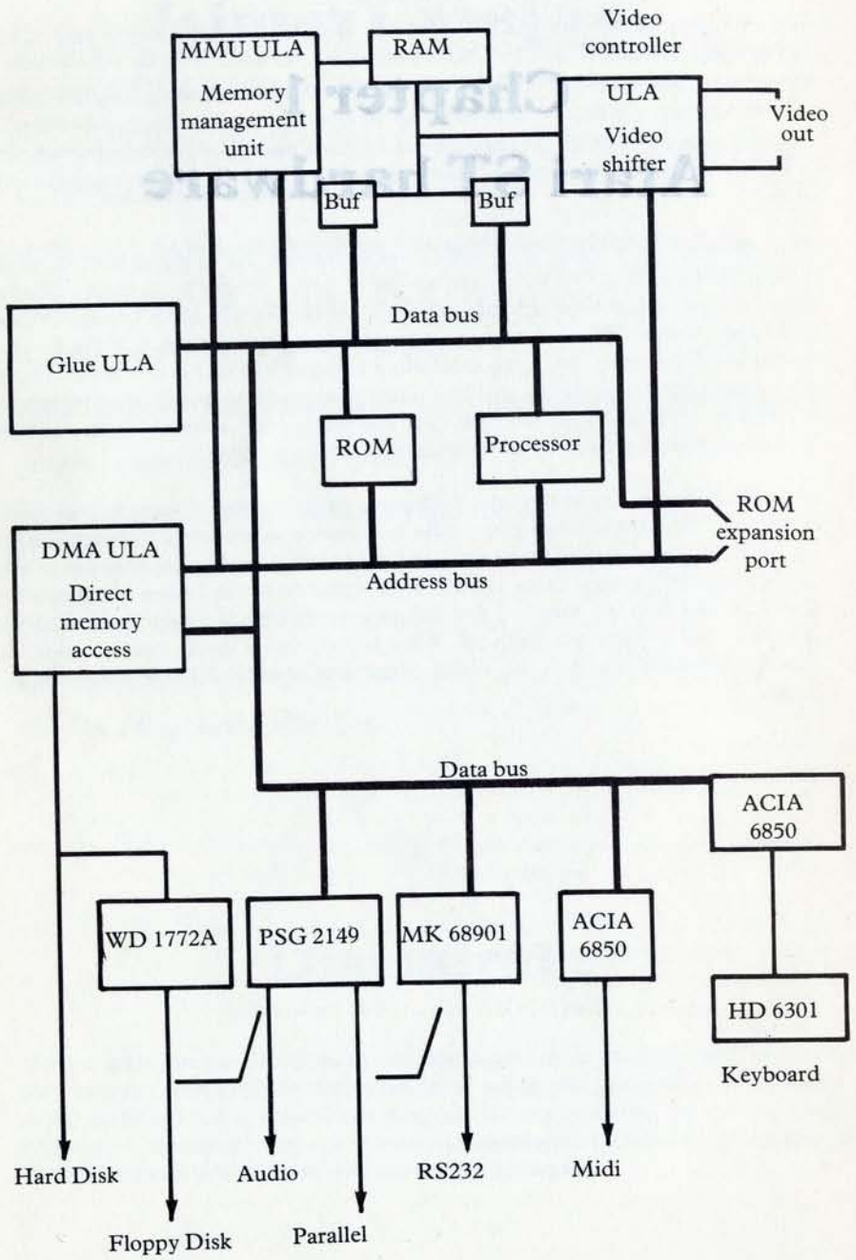
During the past few years, ever since I was responsible for bringing the first mass-produced electronic calculators and then the first true computers to the public at an affordable price, my whole aim has been to bring the benefits of technology to those of average income. We have to get high technology out of the hands of the few into the hands of the many. As I have said before we want 'classes for the masses'. If you give somebody some sophisticated machinery then you'll be surprised what they can do with it. Time and again we have been amazed at what users have done with the technology when it is made freely available at an affordable price.

And that brings me on to this series of books, edited by my old acquaintance Robin Bradbeer. It is impossible to give all the information necessary to completely cover all the uses of a computer in the instruction manual. Also, if more than one person explains something they bring out differing strengths of the system. This series of books should help all users of the ST to get to know the machine better and therefore use it more productively. Who knows, we at Atari may yet again be surprised by what you, the user, can do with the affordable technology that we have provided.

Jack Tramiel
1986

Chapter 1
Atari ST hardware

ATARI ST BLOCK DIAGRAM



General hardware description

The Atari ST computer system consists of a console unit featuring an integral keyboard, a display screen, sound subsystem, peripheral input/output and an operating system. Expansion ports are provided for the connection of a variety of peripheral devices i.e. a mouse, joystick, printer, modem, floppy disk, ROM cartridge application program etc.

The Atari ST console contains an 8MHz MC68000 16 bit microprocessor, at least 512K of resident RAM and a 192K ROM operating system. A Mostek MK68901 multi function peripheral (MFP) device provides the general purpose interrupt control and timers and a single direct main memory access channel, giving both high and low speed access support, through a 32-bit FIFO to the 8 bit device controllers.

User input is via the integral intelligent keyboard, an external mechanical and or optical mouse, or a switch type joystick. The keyboard communicates bidirectionally at 7 Kbits/s with the main unit via a 1MHz HD6301 8-bit microprocessor in the keyboard unit, and a MC6850 asynchronous communications interface adapter (ACIA) in the ST console.

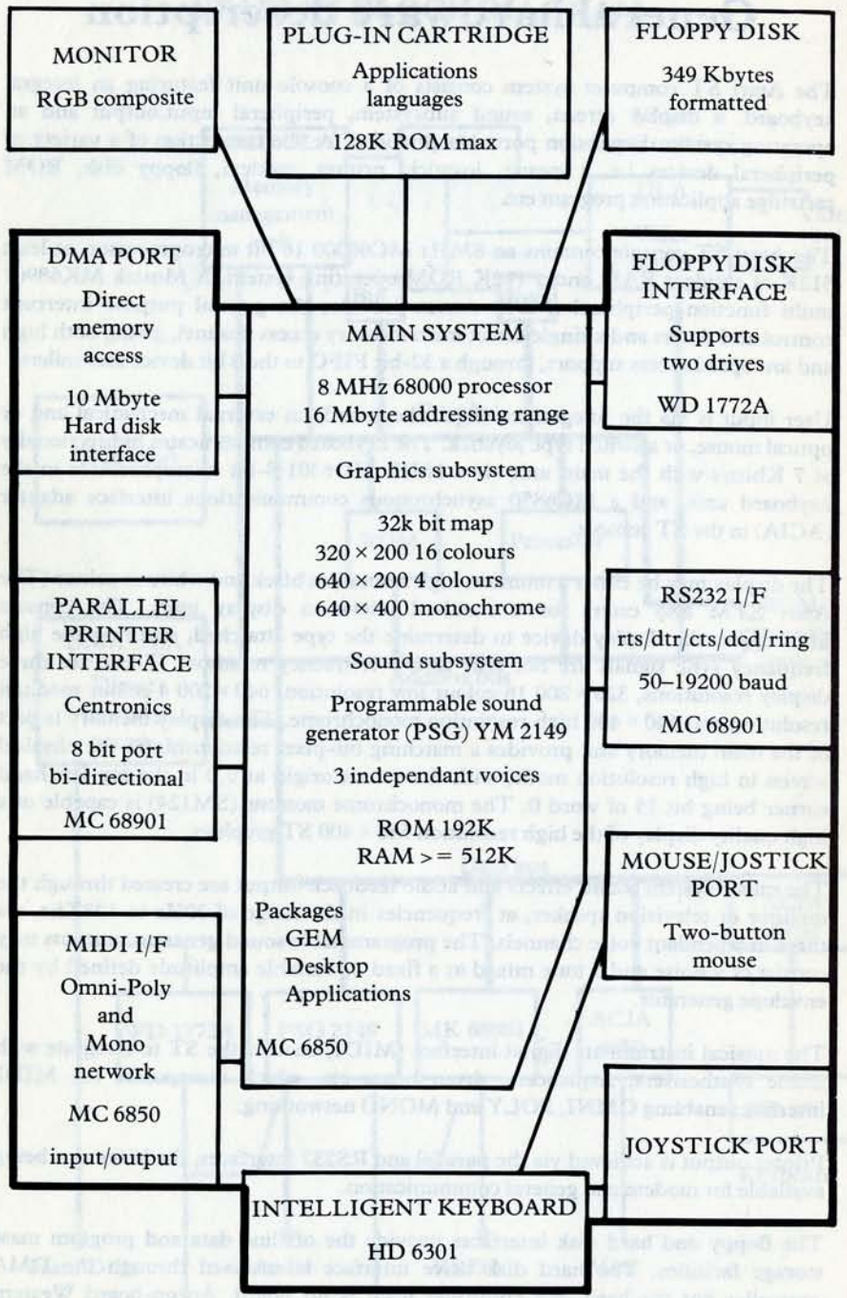
The display may be either a monitor, high resolution black and white or colour (The Atari STM also caters for a standard television display unit). The console interrogates the display device to determine the type attatched, ensuring the high frequency sync signals are not sent to low frequency monitors. There are three display resolutions, 320 × 200 16 colour low resolution, 640 × 200 4 colour medium resolution and 640 × 400 high resolution monochrome. The display memory is part of the main memory and provides a matching bit-pixel relationship to the physical screen in high resolution mode, with the screen origin at 0,0 in the top left hand corner being bit 15 of word 0. The monochrome monitor (SM124) is capable of a high quality display of the high resolution 640 × 400 ST graphics.

The music system sound effects and audio feedback output are created through the monitor or television speaker, at frequencies in the range of 30Hz to 128Khz, via three independant voice channels. The programmable sound generator outputs may consist of a noise and a tone mixed at a fixed or variable amplitude defined by the envelope generator.

The musical instruments digital interface (MIDI) enables the ST to integrate with music synthesisers, sequencers, drum boxes etc. which incorporate the MIDI interface; enabling OMNI, POLY and MONO networking.

Printer output is achieved via the parallel and RS232 interfaces, the latter also being available for modem and general communication.

The floppy and hard disk interfaces provide the off-line data and program mass storage facilities. The hard disk drive interface is accessed through the DMA controller but the hard disk controller itself is off board. An on-board Western Digital WD1772A supplies the floppy disk drive controller which interfaces the Atari ST 3.5" disk drives (SF354 or SF314).



Main system surrounded by device subsystems

The operating system may be either in 192K of ROM, or an image file on disk loaded by the disks boot sector, featuring the GEM operating environment of windows, icons, pull down menus. The ST is also supplied with two language implementations, an interpreted BASIC and Atari LOGO.

The ST can accept other operating systems loaded via the boot sector or brought up by a driver in an 'AUTO' folder.

Atari ST console expansion connections overview

Power
7 way DIN 7P plug
(if external)

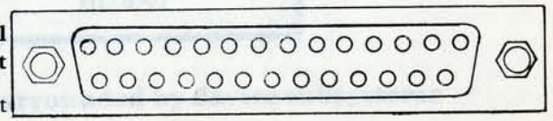
Midi out
5 way DIN 5S socket

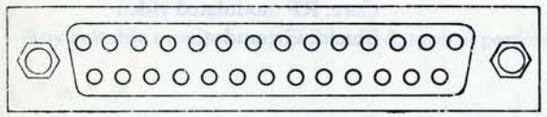
Midi in
5 way DIN 5S socket

Video
RCA pin jack (if fitted)

Monitor output
Composite & RGB
13 way DIN 13S socket

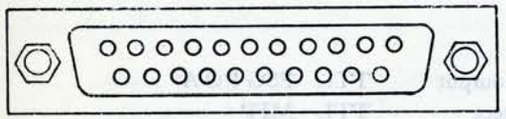
Parallel 8-bit bidirectional printer port
25 way DB 25S socket



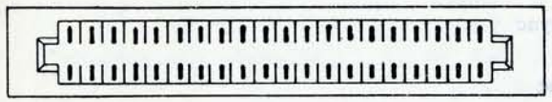


RS232C Modem port
25 way DB 25P plug

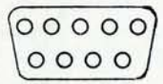
Floppy Disk Controller
(Up to 2 drives)
14 way DIN 14S socket



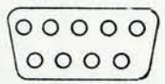
Hard disk port (DMA)
19 way DB 19S socket



ROM expansion port
40 way socket



Joystick
9 way DB 9P plug



Joystick
(configured as a mouse)
9 way DB 9P plug

MONITOR / TV OUTPUT

Monochrome Monitor

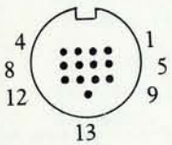
Atari SM124
71.25 Hz scan rate

Colour Monitor

Atari SC1224 RGB
50/60 Hz scan rate

Television *(where fitted)*

RCA pin jack
Core: RF modulated video
Shield: Ground



13 way DIN 13S socket

Sync 5V active low 3.3 Kohm
Audio 1V pk-pk 10 Kohm
Video 1V pk-pk 75 ohm

Pin	*Function*	*ST signal processing device*
1	Audio out	
2	Composite video	
3	General purpose output	TTL PSG I/O A
4	Monochrome detect active low, 1K pull up to 5V	TTL MFP
5	Audio in	
6	Green	
7	Red	
8	Ground	
9	Horizontal sync	
10	Blue	
11	Monochrome	
12	Vertical sync	
13	Ground	

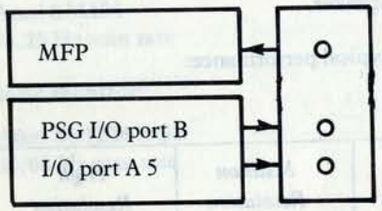
MONITOR OUTPUT

The monitor output supports either a high resolution black and white monitor (Atari SM124) or a medium resolution colour monitor (Atari SC1224). Sound is reproduced through the display device speaker.

Any suitable monitor may be attached, typical performance:

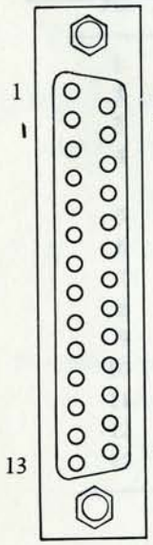
	Low Resolution	*Medium Resolution*	*High Resolution*
Resolution	452 × 585 pixels	653 × 585 pixels	895 × 585 pixels
Video Bandwidth	10 Mhz	18 Mhz	18 Mhz
Slot pitch (typ)	0.64mm	0.41mm	0.31mm
Vertical scan	50/60 Hz	50/60 Hz	71.2 Hz
Input video	1 VDC pk-pk		
audio	1 VDC pk-pk		
Sync	5 VDC active low		

PARALLEL PRINTER INTERFACE



The parallel port interface provides an 8-bit data communication channel controlled by a strobe signal generated by the ST, indicating that data bits are available on the data lines for transfer to the peripheral, and a busy signal generated by the peripheral (usually a printer) indicating either that it is busy, has a fault or possibly out of paper if a printer.

25 way DB 25S socket



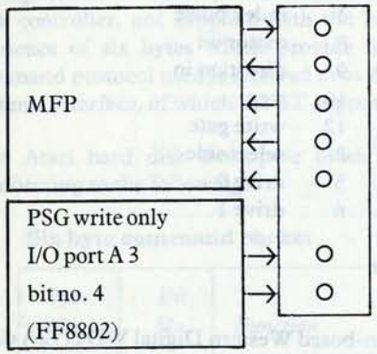
Pin	*Function*	
1	Strobe	
2	Data 1	Data generated at a typical rate of 4kbytes/s by the PSG I/O port B
3	Data 2	
4	Data 3	
5	Data 4	
6	Data 5	
7	Data 6	
8	Data 7	
9	Data 8	
10	n.c	Acknowledge is not supported
11	Busy	
12–17	n.c	
18–25	Ground	

The parallel port strobe signal generated by the PSG I/O port A (pin 1), supplies the data transfer synchronization.

The busy signal (pin 11) is read by the console MFP and provides the handshake control.

The strobe signal is active low, the busy signal active high, with a 1Kohm pull up resistor to +5V. All signals are at TTL levels.

RS232 MODEM INTERFACE

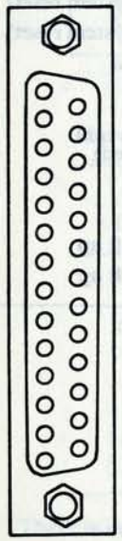


The RS232 interface is controlled via the PSG I/O port A (RTS and DTR) and the MFP (CTS, DCD and RI) transmitting and receiving data within the range 50 to 192K baud, the timing synchronization is generated by the multi-function processor (MFP) timer D.

The interface supports hardware handshake control:

Transmit PSG I/O port A	*Receive MFP inputs*	
RTS	CTS Ring	Software control is through Xon/off protocol.
DTR	DCD	

25 way DB 25P plug

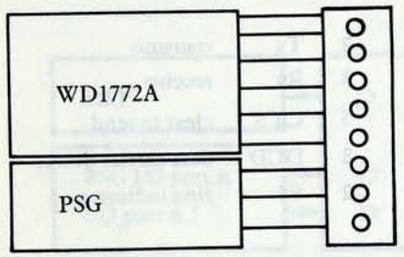


Pin	*Function*	
1	Grd	Protective ground
2	Tx	Transmit data
3	Rx	Receive data
4	RTS	Ready to send
5	CTS	Clear to send
6	n/c	
7	Gnd	Signal ground
8	DCD	Data carrier detect
9–19	n/c	
20	DTR	Data terminal ready
21	n/c	
22	Ri	Ring indicator
23–25	n/c	

RS232 signal levels

Zero +3v to +12v
One –3v to –12v

FLOPPY DISK INTERFACE

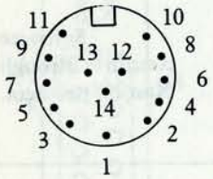


4	index pulse
8	motor on
9	direction in
10	step
12	write gate
2	select side
5	drive 0
6	drive 1

Main console *I/O port*

The floppy disk interface is based on an on-board Western Digital WD1772A disk controller and supports a maximum of two drives. There is no hardware sensing of disk removal. The drives provide fast storage and retrieval of data and programs on 3.5" flexible micro disks.

14 way DIN 14S socket



Pin	*Function*	
1	read data	TTL active low, 1K pull up
2	select side 0	TTL active high (high system reset)
3	logic ground	*pair with read data*
4	index pulse	TTL active low, 1K pull up
5	select drive 0	TTL active low (high system reset)
6	select drive 1	TTL active low (high system reset)
7	logic ground	*pair with write data*
8	motor on	TTL active low
9	direction in	TTL active low
10	step	TTL active low (inverted)
11	write data	TTL active low
12	write gate	TTL active low
13	track 00	TTL active low, 1K pull up
14	write protect	TTL active low, 1K pull up

The shield must not be connected on the ST side.

Data is written to 512 byte sectors.

DIRECT MEMORY ACCESS PORT

This port can be use to provide access to a hard disk or a compact disk. The hard disk controller, not supplied with the basic ST system, communicates through a sequence of six bytes which provide format, read and write facilities etc. The command protocol used is referred to as ANSI X3T9.2, a SCSI-like small computer systems interface, of which the ST supports a small subset.

The Atari hard disk descriptor block consists of a six byte command packet conforming to the following:

Six byte command packet

Byte No.	*Bit No.*	*Function*	*Range*
0	0–4	Operation code	0–31
	5–7	Controller number	0–7
1	0–4	Head number	0–31
	5–7	Drive number	0–7
2	0–5	Sector number	0-63
	6–7	Cylinder number high	
3	0–7	Cylinder number low	
4	0–7	Sector count	
5	0–7	Control byte	

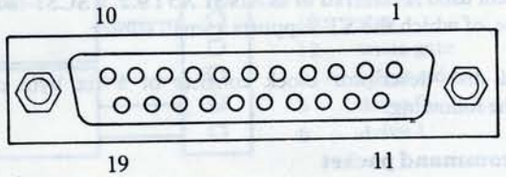
Hard disk command code summary

Op code Dec	*Hex*	*Command*	
5	05	Verify track	Multi-sector
6	06	Format track	transfer
8	08	Read sector	with implied
10	0A	Write sector	seek
11	0B	Seek	
13	0D	Correction pattern	
26	1A	Mode sense	

There is only one DMA channel which is shared by both high- (up to 8 Mbit/s) and low- (250 to 500 Kbit/s) speed 8-bit device controllers.

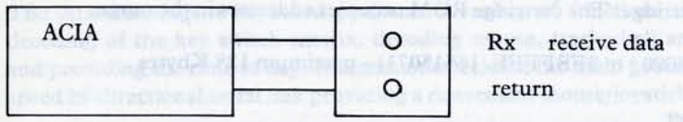
DMA interface port socket

19 way DB 19S socket



Pin	*Function*	*Signal type*
1	data 0	TTL
2	data 1	
3	data 2	
4	data 3	
5	data 4	
6	data 5	
7	data 6	
8	data 7	
9	chip select	TTL active low
10	interrupt request	TTL active low, 1K pull up
11	ground	
12	reset	TTL active low (system reset)
13	ground	
14	acknowledge	TTL active low
15	ground	
16	A1	TTL
17	ground	
18	read/write	TTL
19	data request	TTL active low, 1K pull up

MUSICAL INSTRUMENT INTERFACE (MIDI)



The MIDI interface functions through an MC6850 asynchronous communications interface adaptor (ACIA) whose control/status register is located at $FFC04 (16776196); data is passed in the register at offset 2 from the control/status register.

Data is transmitted serially via the MIDI ports through 2 pins asynchronously using the protocol:

One start bit
Eight data bits
One stop bit
No parity
31.25 Kbaud.

The MIDI ports operate in RS232 current loop mode, that is:

Signal levels:

Zero	5 mA
One	no current

The MIDI OUT port also supports the optional through port which merely provides the MIDI IN signals through an opto-coupled isolator at the MIDI OUT connector.

Control of the port is available through the ST's extended BIOS.

MIDI in

5 way DIN 5S socket

Pin	*Function*
1	n.c
2	n.c
3	n.c
4	In receive data
5	In loop return

MIDI out/thru

5 way DIN 5S socket

Pin	*Function*
1	Through t'mit data
2	Shield ground
3	Through loop return
4	Out transmit data
5	Out loop return

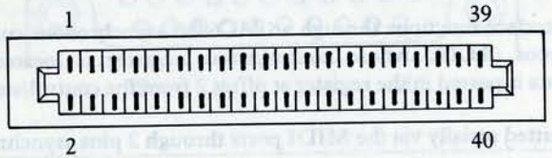
The MIDI ports may be used to network data between connected computers.

PLUG-IN CARTRIDGE PORT

This port provides a plug-in cartridge facility that does not sense in hardware the presence of a cartridge. The cartridge ROM occupies addresses in the range:

$FA0000 (16384000) to $FBFFFF (16515071) – maximum 128 Kbytes

40 way 40S socket



Pin	*Function*	*Pin*	*Function*
1	power +5 Vdc	21	address 8
2	power +5 Vdc	22	address 14
3	data 14	23	address 7
4	data 15	24	address 9
5	data 12	25	address 6
6	data 13	26	address 10
7	data 10	27	address 5
8	data 11	28	address 12
9	data 8	29	address 11
10	data 9	30	address 4
11	data 6	31	ROM3 select
12	data 7	32	address 3
13	data 4	33	ROM4 select
14	data 5	34	address 2
15	data 2	35	upper data strobe
16	data 3	36	address 1
17	data 0	37	lower data strobe
18	data 1	38	ground
19	address 13	39	ground
20	address 15	40	ground

Only the lower 15 address lines are available to the ROM cartridge which does not provide a 'write' line.

Intelligent keyboard (ikbd) interface

The Atari intelligent keyboard performs a variety of functions that include the decoding of the key switch matrix, decoding mouse, trackerball and joystick data and providing the time of day. It communicates with the main processor over a high speed bi-directional serial link providing a convenient mouse/joystick interface.

The keyboard consists of a series of make/break key switches for which the ikbd generates keyboard scan codes for each key press and release, chosen mainly for compatibility with the Digital Research graphic system (GSX). The key codes (see table in appendix D) are defined for the whole range of international keyboards such that each code has a predefined key press meaning, irrespective of the presence of the key switch. The break code for each key is signified by bit 7 of the corresponding make code for the key being set; the codes #$F6 to #$FF are reserved for keyboard system functions.

The keyboard controller contains a 1 MHz HD6301 8-bit microprocessor that communicates with the ST's MC6850 asynchronous communications interface adaptor (ACIA) at a fixed 7.8 Kbit/s. The keyboard not only transmits the encoded key scan codes (with a two key rollover), it also enables the programmer to interrogate the status, define the read rates and sensitivity of the mouse and joysticks under software control.

The time-of-day clock incorporated in the keyboard controller is held to a resolution of 1 second and may be read and set from software. The keyboard may be reset, without affecting the time held by the clock, to its power-up parameters.

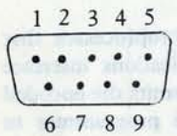
When reset, the keyboard controller performs a simple ROM (checksum), a series of RAM and key (stuck) checks, correct operation is indicated by the return of the version/release number of the ikbd controller.

Mouse/joystick interface

The mouse and joysticks work on the basic unit of an 'event', this is defined as either the opening or closing of a switch, or of motion beyond a predefined programmable threshold level. The mouse is capable of a resolution of 200 events per inch (4 events/mm) and is scanned at such a rate as to permit tracking velocities of up to 10 inches per second (250mm/s), at a maximum pulse phase error of 50%.

Motion, which produces make then break cursor keycodes, can be reported in three different ways: relative, absolute and cursor key motion (motion per keystroke is independently programmable in both axes). The mouse buttons can also be treated as part of the mouse or as additional keyboard keys.

9 way DB 9P plug



Port 0 is configured for mouse operation

Port 1 is the second joystick interface

Pin	*Joystick* Function	*Mouse/Joystick 0* Function
1	Up	XB/Up
2	Down	XA/Down
3	Left	YA/Left
4	Right	YB/Right
5	reserved	n.c
6	Fire	left button/Fire
7	Power	+5v
8	Gnd	Ground
9	n.c	right button/Joystick 1 fire

The mouse unit provides interactive input to programs like the desktop applications, permitting a convenient method of selecting from a menu of facilities shown symbolically as icons or simply as text. Port zero is configured for the mouse, but may also be connected to a joystick.

The joystick is invariably used in games applications; but may also be used instead of the cursor keys, for fine control of the screen cursor position (one pixel movement).

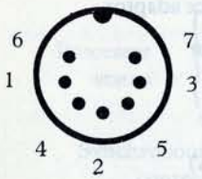
The joystick fire and mouse buttons close to ground.

Power supply

The power supply provides power for the main system board, the keyboard controller, any connected expansion ROM and expansion RAM.

The supply is fused, the levels are regulated for over-voltage and incorporate over-current protection.

7 way DIN 7P plug



Pin	Function
1	+5 VDC
2	n.c
3	Ground
4	+12 VDC
5	–12 VDC
6	+5 VDC
7	Ground

Power levels:

15VDC @ 3A	5%
+12VDC @ 0.03A	10%
–12VDC @ 0.03A	10%

The power supply may be integral with the main unit.

Processor device outlines

MC68000 8 MHz microprocessor

WD1772A floppy disk controller

MK68901 multi-function processor

MC6850 asynchronous communications interface adaptor

YM2149 programmable sound generator

Custom designed devices (ULAs):

Direct memory access controller (DMA)

Memory management unit (MMU)

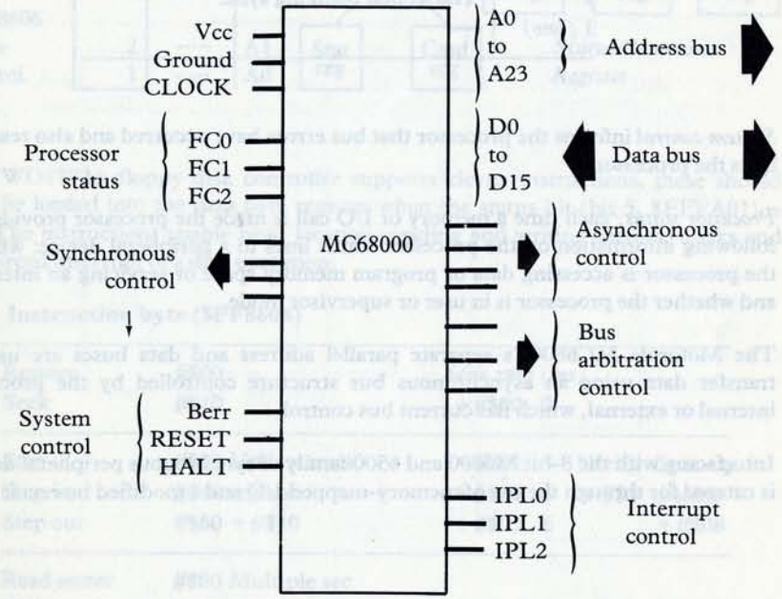
Video controller (Shifter)

General housekeeping (Glue)

MOTOROLA MC68000 MICROPROCESSOR

Signal I/O

The following is a very brief description of the signal I/O of the Motorola MC68000.



A high-density, n-channel, silicon-gate depletion load 16-bit microprocessor in a 64 pin DIL package.

The *address bus* (A0–A23) enables the MC68000 to address 16 megabyte of data or 8 Megaword of instructions. The address bus provides the level being serviced, during an interrupt, on address lines A0 to A3 while A4 to A23 are held high.

The *data bus* (D0–D15) enables the transfer of word and byte-sized chunks of data. During an *interrupt* acknowlege, a vector number may be placed on lines D0 to D7 by a peripheral device.

Bus arbitration control allows a peripheral device to control the MC68000 bus (bus master); any external request will be granted on a priority basis between the competing devices.

Interrupt control provides a priority level from peripherals requesting processor control enabling selection of multiple interrupts on a priority basis. Zero implies that there is no interrupt present and 7 is a non maskable interrupt.

Level	*Autovector*
7 *(high)*	Non maskable interrupt
6	MC68901 multi-function processor
5	—
4	Vertical blanking sync.
3	—
2	Horizontal blanking sync.
1 *(low)*	—

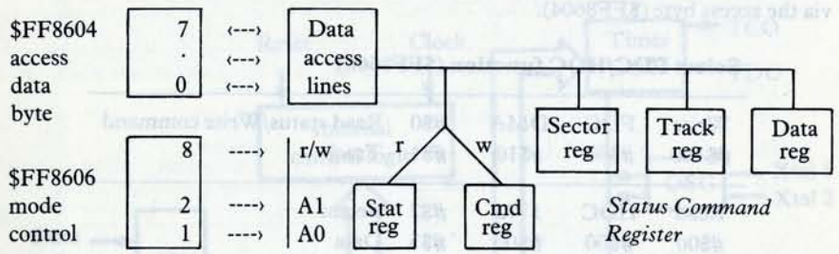
System control informs the processor that bus errors have occurred and also resets or halts the processor.

Processor status: each time a memory or I/O call is made the processor provides the following information on the processor status lines to a peripheral device: whether the processor is accessing data or program memory space or servicing an interrupt; and whether the processor is in user or supervisor mode.

The Motorola MC68000's separate parallel address and data buses are used to transfer data using an asynchronous bus structure controlled by the processor, internal or external, which has current bus control.

Interfacing with the 8-bit M6800 and 6500 family of synchronous peripheral devices is catered for through the use of memory-mapped I/O and a modified bus cycle.

WD1772A FLOPPY DISK CONTROLLER



The WD1772A floppy disk controller supports eleven instructions, these should only be loaded into the data byte register when the status bit (bit 5, $FFFA01) is off. The instructions enable head location, reading and writing sectors, tracks and the forced interrupt of a disk operation:

Instruction byte ($FF8604)

Restore	#$00	Seek rate *(ms)*			
Seek	#$10	+#$00	2		
Step	#$20 Update track	+#$01	3	Verify	Spin-up
Step in	#$40 register	+#$02	5	+#$4	disable
Step out	#$60 +#$10	+#$03	6		+#$08
Read sector	#$80 Multiple sec				
Write sector *	#$A0 +#$10				
Read address	#$C0 Add 30 ms delay				
Read track	#$E0 +#$04				
Write track *	#$F0				
Force interrupt	#$D0	+#$00 End with no interrupt			
		+#$04 Interrupt on index pulse			
		+#$08 Immediate interrupt			

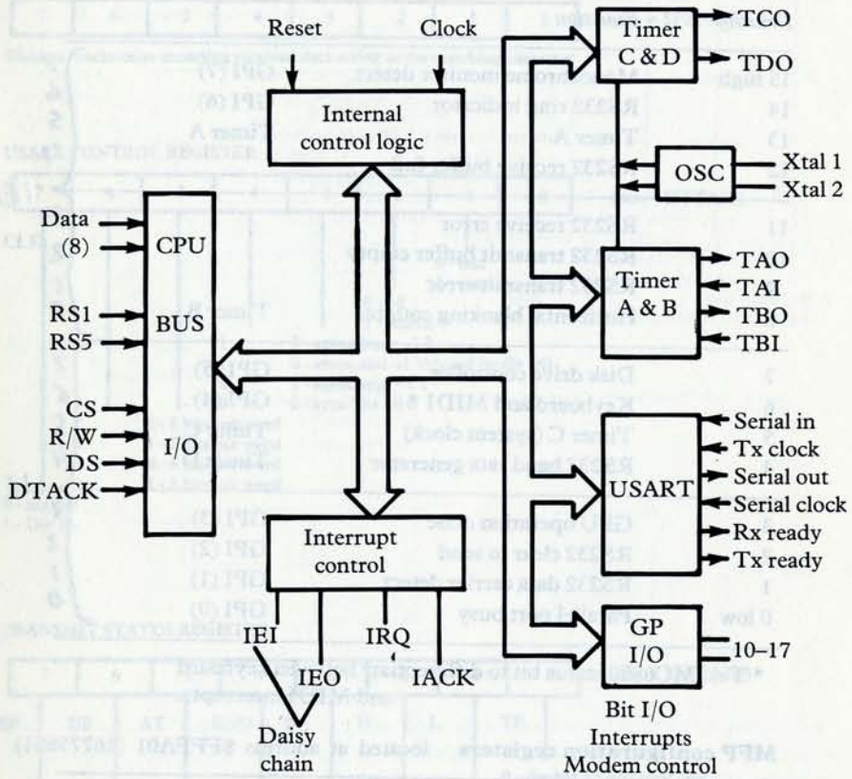
* May contain +#$02: Write precompensation disabled.
Write sector may also contain +#$01: Write deleted data mark.

Commands are passed to the FDC (and an external HDC), by selecting the appropriate FDC or HDC function (Read status/write command, sector, track or data) through the configuration register ($FF8606) and sending instructions or data via the access byte ($FF8604).

Select FDC/HDC function ($FF8606)

Write	FDC	DMA	#$0	Read status/Write command
#$100	#$80	#$10	#$1	Track
Read	HDC	1772	#$2	Sector
#$00	#$00	#$00	#$3	Data

MC68901 MULTI-FUNCTION PROCESSOR



The MC68901 contains a single channel USART capable of operating in full duplex, at a rate of 62.5Kb/s asynchronous, 1Mb/s synchronous from an internal or external Baud rate generator. The USART also supports DMA handshake signals and modem control.

There are four timers with independant operation and vectored interrupts, the timers have the following preferred timer uses:

Timer A: Stand alone applications and independent software vendor.
Timer B: Primarily Screen Graphics (hblank, sync etc.)
Timer C: System timing (GSX, GEM, Desktop, etc). Suitable for delays and general timing applications (200Hz).
Timer D: RS 232 port baud rate control.

Eight individually programmable I/O pins with interrupt capabilities are also available.

MC68901 interrupt control

MFP hardware bound interrupts

Priority	*Function*		
15 high	Monochrome monitor detect	GPI (7)	7
14	RS232 ring indicator	GPI (6)	6
13	Timer A	Timer A	5
12	RS232 receive buffer full		4
11	RS232 receive error		3
10	RS232 transmit buffer empty		2
9	RS232 transmit error		1
8	Horizontal blanking counter	Timer B	0
7	Disk drive controller	GPI (5)	7
6	Keyboard and MIDI *	GPI (4)	6
5	Timer C (system clock)	Timer C	5
4	RS232 baud rata generator	Timer D	4
3	GPU operation done	GPI (3)	3
2	RS232 clear to send	GPI (2)	2
1	RS232 data carrier detect	GPI (1)	1
0 low	Parallel port busy	GPI (0)	0

IERA

IERB

* Test MC6850 status bit to differentiate between keyboard and MIDI interrupts.

MFP configuration registers located at address $FFFA01 (16775681)

Offset Dec	*Offset* Hex	*Function*	*Offset* Dec	*Offset* Hex	*Function*
1	01	General purpose I/O	25	19	Timer A control
3	03	Active edge	27	1B	Timer B control
5	05	Data direction	29	1D	Timers C & D control
7	07	Interrupt enable A	31	1F	Timer A data
9	09	Interrupt enable B	33	21	Timer B data
11	0B	Interrupt pending A	35	23	Timer C data
13	0D	Interrupt pending B	37	25	Timer D data
15	0F	Interrupt in-service A	39	27	Sync character
17	11	Interrupt in-service B	41	29	Usart control
19	13	Interrupt mask A	43	2B	Receiver status
21	15	Interrupt mask B	45	2D	Transmitter status
23	17	Vector base address	47	2F	Usart data

The MC68901 usart registers are accessible from Extended BIOS.

SYNCHRONOUS CHARACTER REGISTER

7	6	5	4	3	2	1	0

(scr = $FFFA27)

Used to synchronize incoming received data acting as the matching character

USART CONTROL REGISTER

7	6	5	4	3	2	1	0

(ucr = $FFFA29)

CLK

0 - normal
1 - Div 16

0 - 8 bits per word
1 - 7 bits per word
0 - 6 bits per word
1 - 5 bits per word

1 - async Start -›1 2
0 - async and -›1 ½ (used by div 16)
1 - async stop -›1 1
0 - sync bits -›0 0

0 - off
1 - enable

0 - odd
1 - even

TRANSMIT STATUS REGISTER

7	6	5	4	3	2	1	0
BE	UE	AT	END	B	H	L	TE

(tsr = $FFFA2D)

0 - Tx buffer read 9
1 - Tx word transferred to Tx shift register

0 - Tx status register read 10
1 - word transmitted and Tx buffer empty

0 - Disable Tx
1 - Enable Rx when Tx disabled after last char sent

0 - Tx enabled 9
1 - Tx disabled after last char sent

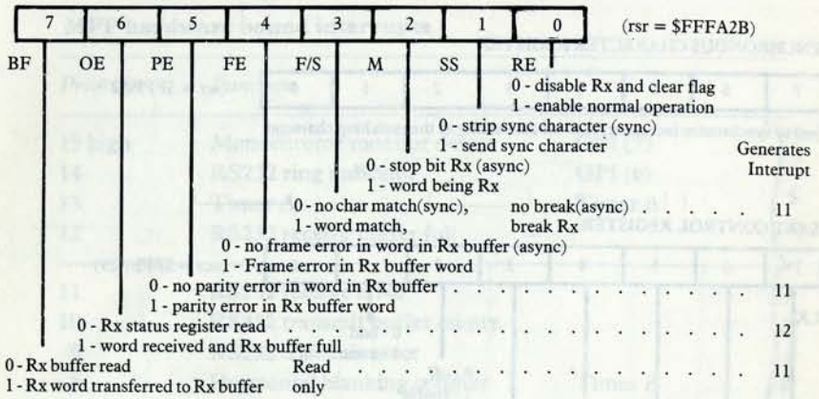
0 - normal Tx
1 - Send a break

0 - high imp
1 - low
0 - high
1 - loopback (connect o/p to i/p)
async

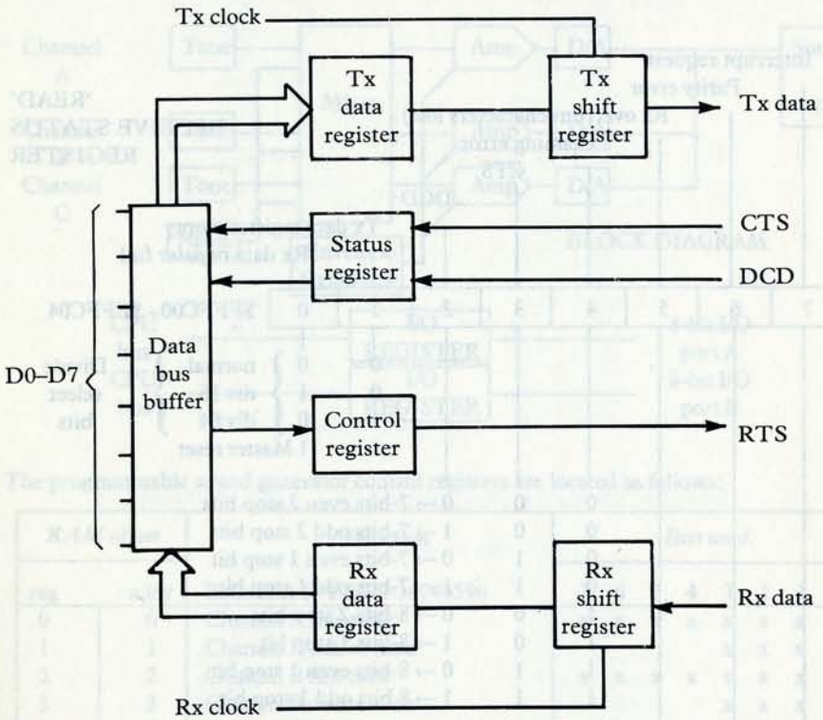
Configure Tx o/p when Tx disabled.

0 - disable Tx and clear flag
1 - enable normal operation

RECEIVE STATUS REGISTER



MC6850 ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTOR



The MC6850 ACIA provides data formatting and control of a serial interface to an 8-bit bidirectional data bus. At the bus interface, the four ACIA registers, the status and receive data (read only) and the control and transmit data (write only) registers, appear as two addressable memory locations.

The programmable ACIA control register, which sets the format of the serial link, is located at $FFFC00 (16776192) for the intelligent keyboard serial communications link, and at $FFFC04 (16776196) for the MIDI interface.

The ACIA supports peripheral/modem control through:

RTS request to send,
CTS clear to send, and
DCD data carrier detect.

Protocols for 8 and 9 bit transmission using an optional odd or even parity, and one or two stop bits, are available through the programmable control register.

The MIDI port may be configured for a second serial port, but the intelligent keyboard interface is not accessible.

ACIA CONTROL/STATUS REGISTER

Interrupt request
Parity error
Rx overrun (characters lost)
Framing error
CTS
DCD
Tx data register empty
Rx data register full

'READ' RECEIVE STATUS REGISTER

7	6	5	4	3	2	1	0

$FFFC00 - $FFFC04

0	0	normal	Divide select bits
0	1	div 16	
1	0	div 64	
1	1	Master reset	

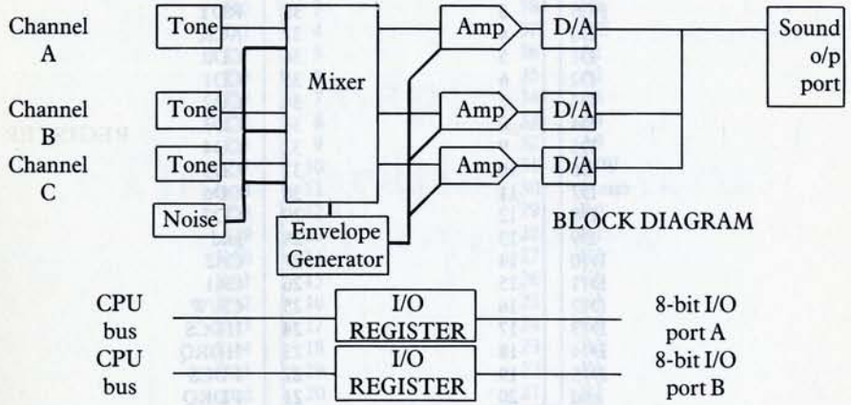
0	0	0 --› 7-bits even 2 stop bits
0	0	1 --› 7-bits odd 2 stop bits
0	1	0 --› 7-bits even 1 stop bit
0	1	1 --› 7-bits odd 1 stop bit
1	0	0 --› 8-bits 2 stop bits
1	0	1 --› 8-bits 1 stop bit
1	1	0 --› 8-bits even 1 stop bit
1	1	1 --› 8 bits odd 1 stop bit

'WRITE' TRANSMIT CONTROL REGISTER

0	0 RTS low Tx interrupt disable
0	1 RTS low Tx interrupt enabled
1	0 RTS high Tx interrupt disable
1	1 RTS low Tx a break onto data output. Tx interrupt disable.

0_Rx interrupt disable
1_Rx interrupt enable (Rx data register full overrun. DCD low-to-high transition)

YM2149 YAMAHA PROGRAMMABLE SOUND GENERATOR



The programmable sound generator control registers are located as follows:

RAM offset		*Function*	*Bits used*							
reg	addr	Base addr $FF8800-16746596	7	6	5	4	3	2	1	0
0	0	Channel A fine tune	x	x	x	x	x	x	x	x
1	1	Channel A coarse tune					x	x	x	x
2	2	Channel B fine tune	x	x	x	x	x	x	x	x
3	3	Channel B coarse tune					x	x	x	x
4	4	Channel C fine tune	x	x	x	x	x	x	x	x
5	5	Channel C coarse tune					x	x	x	x
6	6	Noise period				x	x	x	x	x
7	7	Mixer control–I/O enable	I/O noise tone							
		Fixed amplitude								
8	8	Channel A amplitude				M	x	x	x	x
9	9	Channel B amplitude				M	x	x	x	x
10	A	Channel C amplitude				M	x	x	x	x
		Variable amplitude								
11	B	Envelope period fine	x	x	x	x	x	x	x	x
12	C	Envelope period coarse	x	x	x	x	x	x	x	x
13	D	Envelope shape					C	R	A	H
14	E	I/O port A (output only)								
15	F	I/O port B (centronics)	data							

M = mode fixed/variable. C = cycle. A = alternate. x = bits used. R = ramp. H = hold

DIRECT MEMORY ACCESS CONTROLLER (DMA)

R/W	1	40	+5v
A1	2	39	clk 8Mhz
FCS	3	38	RDY
D0	4	37	ACK
D1	5	36	CD0
D2	6	35	CD1
D3	7	34	CD2
D4	8	33	CD3
D5	9	32	CD4
D6	10	31	CD5
D7	11	30	CD6
D8	12	29	CD7
D9	13	28	gnd
D10	14	27	CA2
D11	15	26	CA1
D12	16	25	CR/W
D13	17	24	HDCS
D14	18	23	HDRQ
D15	19	22	FDCS
gnd	20	21	FDRQ

MEMORY MANAGEMENT UNIT (MMU)

D4	1	68	D3
D5	2	67	D2
D6	3	66	D1
D7	4	65	D0
16Mhz clk	5	64	MAD9
CASOM	6	63	MAD8
CASOL	7	62	MAD7
RASO	8	61	gnd B
latch	9	60	MAD6
+5v	10	59	MAD0
A16	11	58	MAD1
A17	12	57	MAD2
A18	13	56	MAD3
A19	14	55	MAD4
A20	15	54	MAD5
A21	16	53	dtack
LDS	17	52	DE
RASI	18	51	vsync
4Mhz clk	19	50	A1
8Mhz clk	20	49	A2
CASIL	21	48	A3
CASIM	22	47	A4
W/E	23	46	A5
DMA	24	45	A6
WDAT	25	44	+5v
UDS	26	43	A7
gnd A	27	42	A8
CPMCS	28	41	A9
DCYC	29	40	A10
RDAT	30	39	A11
DEV	31	38	A12
AS	32	37	A13
RAM	33	36	A14
R/W	34	35	A15

VIDEO CONTROLLER (SHIFTER)

XTL0	1	40	+5v
32Mhz XTL1	2	39	clk 16Mhz
D0	3	38	CS
D1	4	37	DE
D2	5	36	A1
D3	6	35	A2
D4	7	34	A3
D5	8	33	A4
D6	9	32	A5
D7	10	31	R/W
load	11	30	Mono
D8	12	29	R0
D9	13	28	R1
D10	14	27	R2
D11	15	26	G0
D12	16	25	G1
D13	17	24	G2
D14	18	23	B0
D15	19	22	B1
gnd	20	21	B2

GENERAL HOUSEKEEPING (GLUE)

+5v	1	68	A13
A14	2	67	A12
A15	3	66	A11
A16	4	65	A10
A17	5	64	A9
A18	6	63	A8
A19	7	62	A7
A20	8	61	A6
A21	9	60	A5
A22	10	59	A4
A23	11	58	A3
AS	12	57	A2
FC2	13	56	A1
FC1	14	55	R/W
FC0	15	54	clk 2Mhz
VMA	16	53	SNDCS
ROM4	17	52	gnd
ROM3	18	51	MFPCS
ROM2	19	50	IACK
ROM1	20	49	D1
ROM0	21	48	D0
reset	22	47	UDS
RAM	23	46	LDS
DMA	24	45	BGO
DEV	25	44	MFPINT
FCS	26	43	clk 500Khz
BGI	27	42	6850CS
RDY	28	41	BGACK
VPA	29	40	BR
Berr	30	39	DE
dtack	31	38	vsync
IPL1	32	37	hsync
IPL2	33	36	blank
8Mhz clk	34	35	gnd

Chapter 2

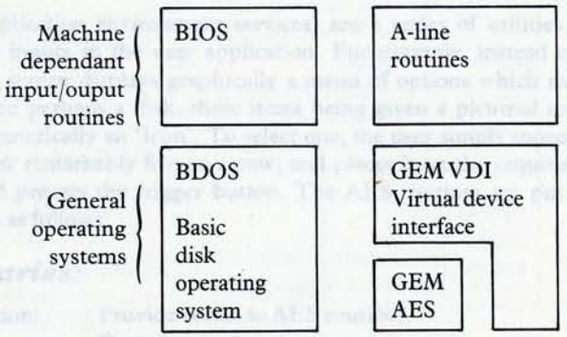
The operating system (TOS) overview

Operating system overview

The Atari ST operating system is in many ways functionally similar to CP/M 68K, with extensions for handling a mouse, sound, the midi interface, an intelligent keyboard and joysticks. A graphics environment manager (GEM) provides additional single-user support for windows and communications via VDI and AES extensions, which accomodate graphics and an applications environment. Program transportability is maintained by splitting the operating systems into machine independant (BDOS, VDI and AES) and machine dependant basic input/output utilities (BIOS and A-line routines).

The ST programmer is given access to the VDI primitives via the A-line routines for much greater graphic application speed.

The disk operating system (DOS) enables routines to access the disc drives with support for existing single user programs, file locking to ensure safe updating, read only and unlock facilities. The access manager improves the file handling with a library of routines that permit indexed files through a tree structure to ensure high operational efficiency. Other functions available are; both data and indexed file system initialization update, duplicate keys and an index file search. Runtime errors may be handled internally, or there is the option to stop and correct the error.



Programmable segments of TOS

The application environment (AES) multitasks using a timeslicing technique and supports a database file management system, real time data aquisition, communications and process control.

The virtual device interface (VDI) allows the use of peripheral independant device drivers and provides a high degree of assistance for advanced user interfaces.

BASIC INPUT/OUTPUT SYSTEM (BIOS)

The BIOS consists of all the machine dependent I/O routines of Digital Research's GEM and additionally provides access to the A-line routines for fast graphics. The I/O functions can be categorized as follows:

GEM BIOS:
- System I/O:
  - Parameter block initialization
  - Console I/O: Data I/O & query
  - Disk I/O: Memory/disk transfers

Atari ST extended BIOS:
- Port I/O: Configure RS232, mouse, midi & sound port
- Screen I/O: Get screen parameters
- Disk I/O: Memory/disk transfers
- Keyboard I/O: Keyboard communications

A-line routines:
- Pixel graphics
- Line graphics
- Sprite graphics
- Bit block transfer
- Mouse handler

BASIC DISK OPERATING SYSTEM (BDOS)

The disk operating systems permits the machine independent routines to access the disk drives and handle file management through the following functions:

- Set/get time and date
- Tree directory management
- File attribute management
- Create/open/close files and disk transfers.

Virtual device interface

The VDI provides a set of graphic function calls that allow portability across physical hardware. Not all the standard VDI calls are implemented on the ST, the VDI tables Chapter 3 are annotated to show those that are missing.

Control I/O:	Initialize graphics & set defaults.
Graphics I/O:	Primitives, lines, polygons, bars, arcs & pies.
Attribute I/O:	Set colour and style.
Raster I/O:	Bit block transfers, fill, font and cursor forms.
Input I/O:	Keyboard/mouse interaction with console.
Inquire I/O:	Get attributes, resolution, style etc.
Special I/O:	Permits specialized functions to be performed.

The AES (application environment services) are a series of utilities that handle graphic based inputs to the user application. For example, instead of asking for INPUT – the screen displays graphically a menu of options which may include a clock, a file and perhaps a disk, these items being given a pictorial representation that is called generically an 'Icon'. To select one, the user simply moves the cursor, which may look remarkably like an arrow, and places it on the required icon using the mouse and presses the trigger button. The AES routines are put into groups called libraries as follows:

AES Libraries:

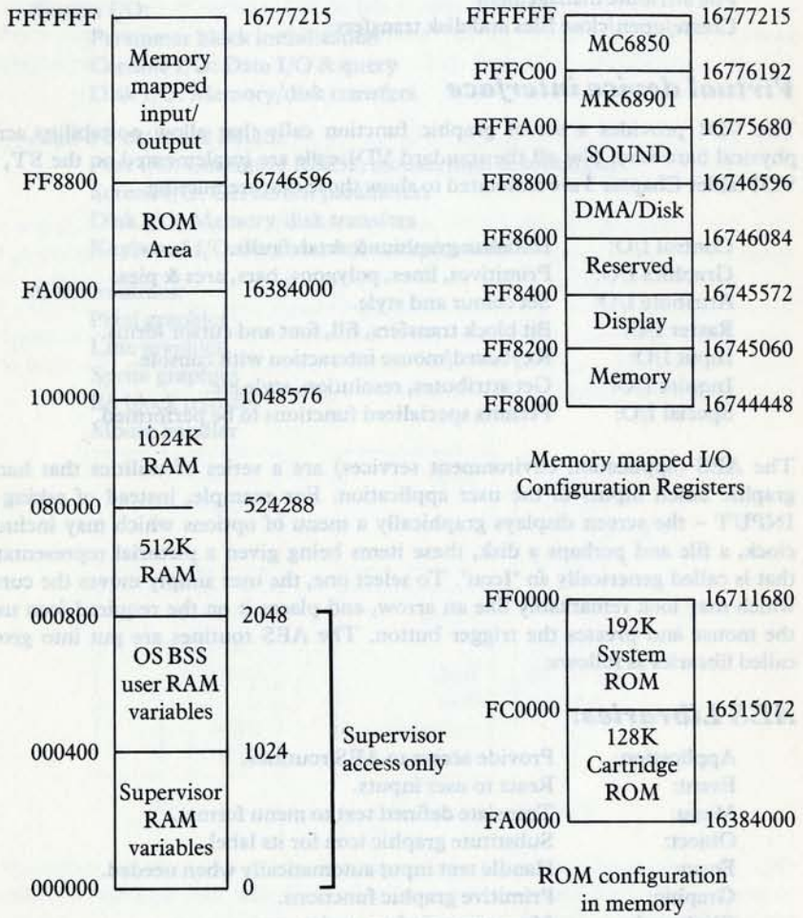
Application:	Provide access to AES routines.
Event:	React to user inputs.
Menu:	Translate defined text to menu format.
Object:	Substitute graphic icon for its label.
Form:	Handle text input automatically when needed.
Graphic:	Primitive graphic functions.
Clipboard:	Management of cut and paste.
File Selector:	Creation/display of user selected file.
Window:	Handle windowing of queried input responses.
Resource:	Interface device dependant drivers to applications.

Application programs

The desktop application provides the support for windowing, control and synchronization of asynchronous events, interprocess communications, loadable device drivers and device independant graphics support. The clock application date stamps each file on creation or modification.

MEMORY ALLOCATIONS

ATARI MEMORY MAP



References to the bottom 2K of memory and the I/O space are classed as supervisor references and attempted access from user mode will cause an error exception trap.

THE SCREEN RAM TAKES THE TOP 32K OF RAM

SYSTEM TABLES

$800			Operating system block storage segment
			Supervisor space
$400		1024	System parameters and variables
$200		512	reserved for OEMs
$100		256	MFP vectored interrupts
		192	
$0BC	Trap #15 vector	188	
$0B8	Trap #14 vector	184	Atari ST BIOS extensions
$0B4	Trap #13 vector	180	GEMBIOS
	Trap #12 vector	176	
$08C	Trap #3 vector	140	
$088	Trap #2 vector	136	BDOS
$084	Trap #1 vector	132	GEMDOS interface
$080	Trap #0 vector	128	
	Int level 7	124	Non maskable interrupt (EG HARDWARE RESET)
	Int level 6	120	MK68901 MFP
	Int level 5	116	
$070	Int level 4	112	Vertical blank sync
$06C	Int level 3	108	Normal interrupt level
$068	Int level 2	104	Horizontal blank sync
$064	Int level 1	100	
$060	Spurious Int	96	
			Unused vectors point to system critical error handler.
$03C	Uninit int vector	60	
		48	
$02C	Emulation 1111	44	Used by some AES functions
$028	Emulation 1010	40	A-line routines entry
$024	Trace	36	
$020	Privilege violation	32	The system variables are in
$01C	Trap instruction	28	the supervisor space and can
$018	CHK instruction	24	be accessed only in supervisor
$014	Zero divide	20	mode.
$010	Illegal instruction	16	
$00C	Address error	12	
$008	Bus error	8	
	Initialise PC		
$000	Reset init SSP	0	

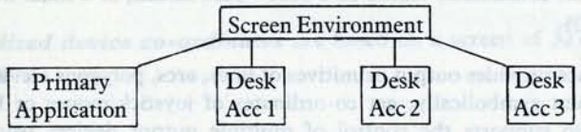
CONFIGURATION REGISTERS

			Functions controlled
FFFFFF	ACIA	16777215	Keyboard and MIDI I/O
FFFC00	MFP	16776192	System clocks System interrupts
FFFA00	Sound	16775680	PSG 3 channel sound, noise tone, amplitude and envelope RS232 Parallel port Disk write
FF8800	DMA/disk	16746596	Floppy/hard disk DMA
FF8600	Reserved	16746084	
FF8400	Display	16745572	Video address Field rate Video mode & pallette
FF8200	Memory	16745060	Memory size
FF8000		16744448	

Resource management overview

The pseudo multitasking kernal can support one primary application and one of three desk accessory programs.

The main application may be GEM or DOS such as GEM desktop application or a word processing package etc.



A minimum space allocation of 128K.

A desk accessory is an application that does not take over the entire display screen, running in a specially designed window. The calculator is a typical accessory.

Only one desk accessory program may be active at a time, and will only load if at least 128K of RAM is left for the primary application.

CPU resources

The dispatcher divides CPU time between primary applications and background processes. These jobs are put into lists; 'Ready for processing' and 'Not ready', and are serviced on a round robin schedule with the current process at the head of the list running. Not ready processes may be waiting for a key press, mouse movement or trigger, time lapse etc.

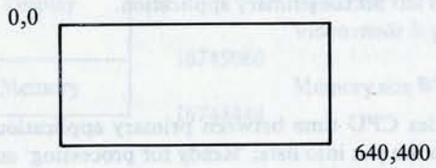
Graphics Concept Overview

The Atari ST graphics is supported at a primitive level through the A-line routines and at a higher level through a limited version of the Digital Research graphic system extension (GSX), which is based on the ANSI virtual device interface (VDI). VDI provides a set of graphic primitives (GDOS) and a library of device drivers (GIOS) for the preparation of transportable software. The whole of GDOS does not form part of the ST operating system and there is no support for 'normalized device coordinates' based on a 32K × 32K screen, or a small number of the VDI functions.

The VDI interface provides output primitives of lines, arcs, polygons etc. and input primitives to point symbolically, get co-ordinates of joystick/mouse or keyboard input etc. It also supports the control of multiple output devices using raster screens.

The A-line routines give very fast access to the primitive pixel, line, sprite and bit block transfer graphic functions at the expense of portability

The Raster co-ordinate graphics are based on screen pixels:



GEM programs are portable but must take into account two possible problem areas:

Screen aspect ratio: Different hardware systems and displays (screen, printer, plotter or another computer) may have different aspect ratios. Producing similar screen designs requires the programmer to scale the data sent to the display device using the aspect ratio returned from the open workstation call.

Language implementations: Different language implementations of a program will require different length text strings to be fitted into windows. The inquire character cell width call in conjunction with the window size returned by the `wind_get` call will enable the programmer to determine the number of characters acceptable.

Alert and Dialog boxes have predetermined responses set up using the resource construction set and therefore do not present a language problem.

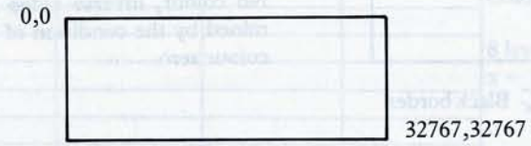
The missing part of GDOS is available as part of the code supplied in certain Digital Research products and may at a later date become more generally available for the ST. On this premise, the details of the missing parts are given coupled to a rider that theyt are not available on the basic system.

GEM usually provides two graphic coordinate systems to the programmer, raster and normalized.

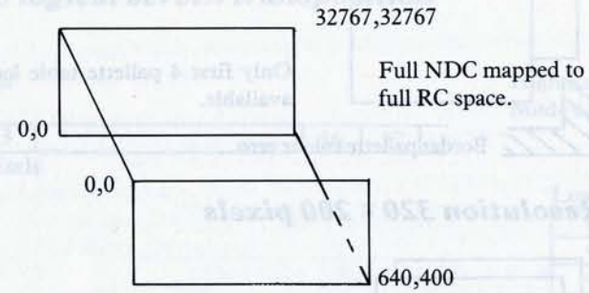
Raster is based on the computer's screen resolution, in the case of the Atari ST 600 × 400 pixels.

Normalized is based on a notional screen of 32767 × 32767 points, the points being translated to the actual screen of the target system by one of the GIOS device drivers. The idea behind this is to write software independant of specific screen resolutions.

Normalized device co-ordinates are based on a screen of 32767 × 32767 pixel dimensions.



Graphic Co-ordinate Computation

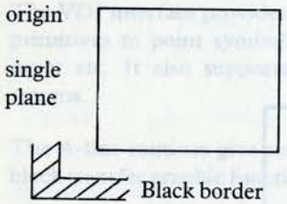


OVERVIEW OF SCREENS

The Atari ST screen may be operated in three different resolution modes, the colours may be chosen from a pallette of 512 colours:

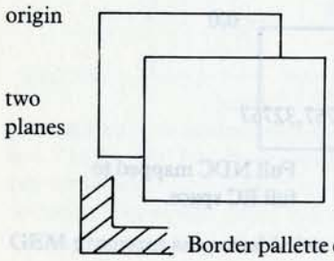
High: 640 × 400 pixel, black and white display
Medium: 640 × 200 pixel, 4 colour display
Low: 320 × 200 pixel, 16 colour display

High Resolution 640 × 400 pixels



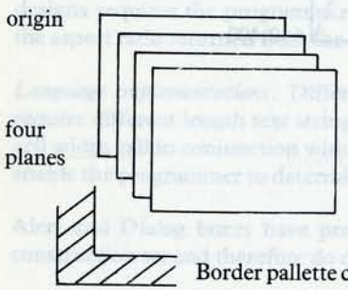
No colour, inverse video is available determined by the condition of bit zero of pallette colour zero.

Medium Resolution 640 × 200 pixels



Only first 4 pallette table lookup entries are available.

Low Resolution 320 × 200 pixels

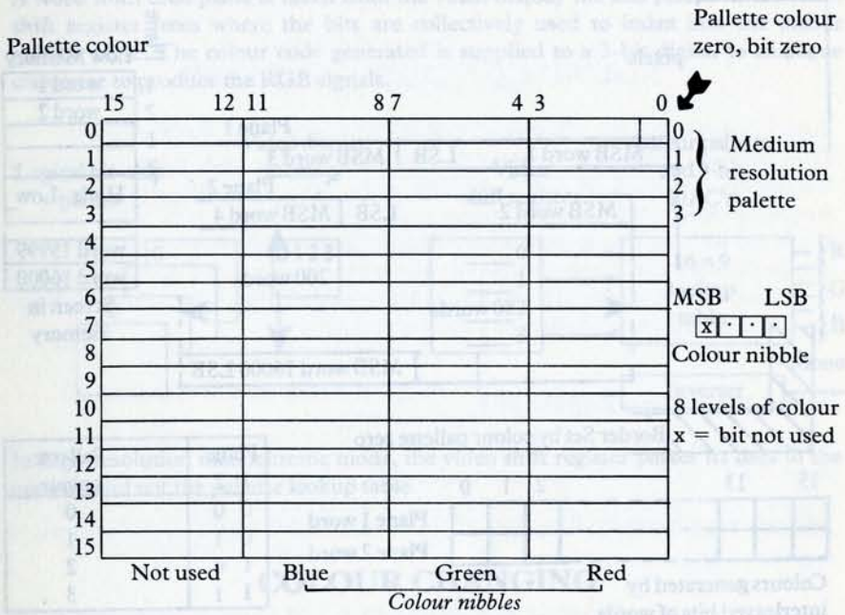


16 word lookup pallette table 9 bits per entry.

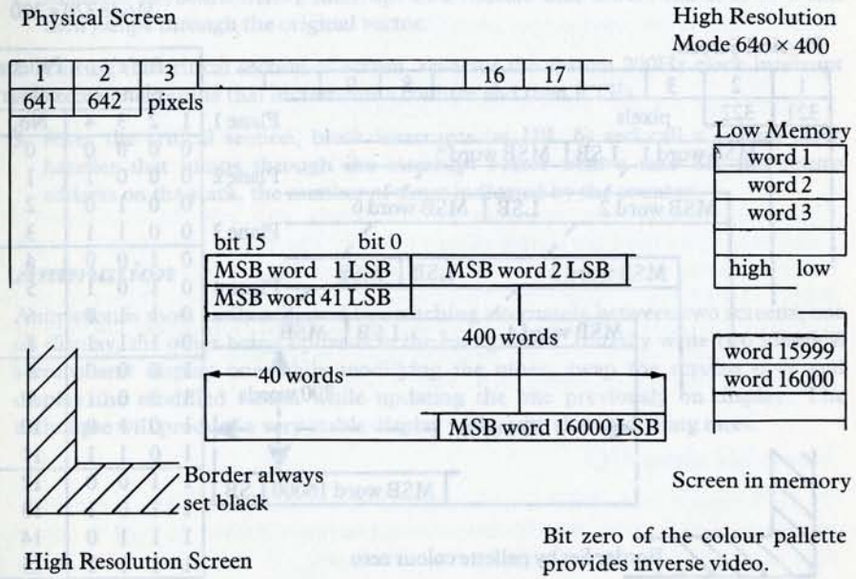
3 red, 3 green and 3 blue on low nibble boundaries, giving 8 × 8 × 8 possible colours. (512)

It is not possible to change resolution while using GEM

Colour Pallette Table

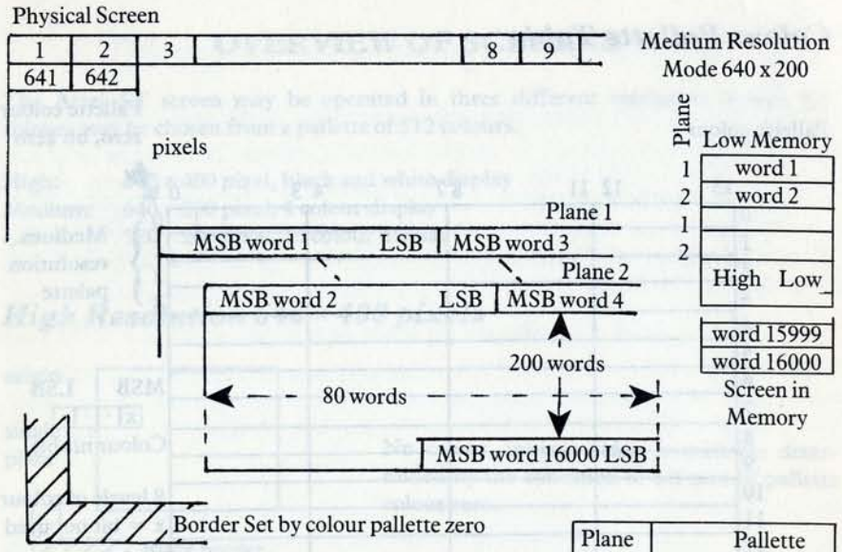


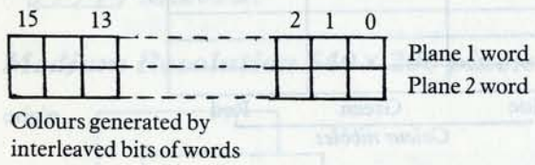
Physical to logical screen transposition



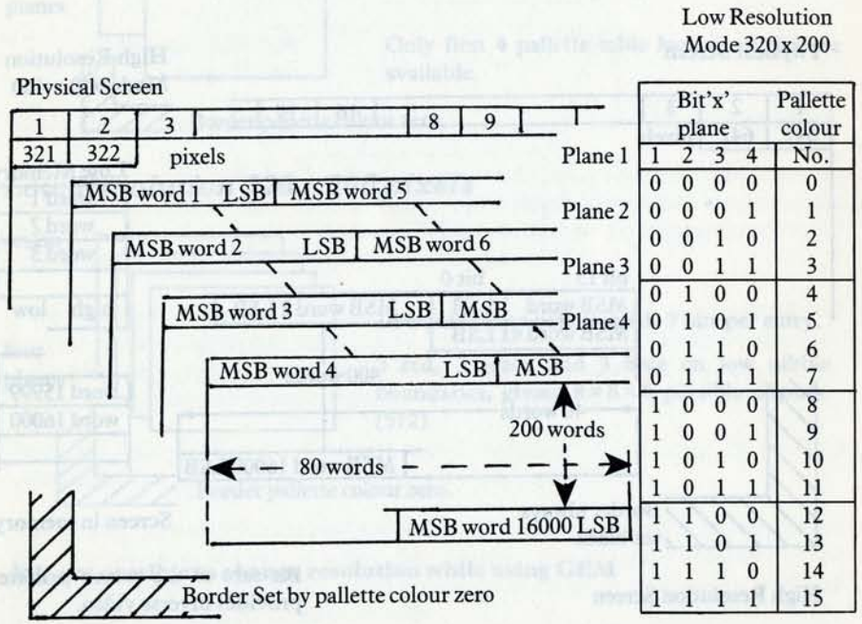
High Resolution Screen

Bit zero of the colour pallette provides inverse video.





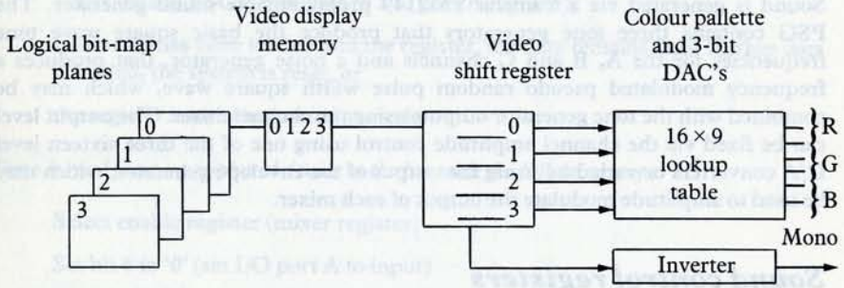
Plane 1	Plane 2	Pallette colour
0	0	0
0	1	1
1	0	2
1	1	3



Bit'x' plane 1	2	3	4	Pallette colour No.
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Colour generation

A word from each plane is taken from the video display file and placed in the video shift register from where the bits are collectively used to index into the colour pallette table. The colour code generated is supplied to a 3-bit digital to analogue convertor to produce the RGB signals.



In high resolution monochrome mode, the video shift register passes its data to the inverter and not the pallette lookup table.

COLOUR CHANGING

To prevent jitter when changing colors using the `Hblank` ($068) and `Hsync` ($120) interrupt vectors, programmers should use the following procedure:

1 Revector keyboard/MIDI interrupt to a routine that lowers the IPL to 5 and then jumps through the original vector.

2 During the critical section of screen, revector the system 200Hz clock interrupt vector to a routine that increments a counter and then RTEs.

3 After the critical section, block interrupts (at IPL 6) and call a system clock handler that jumps through the interrupt vector with a fake SR and return address on the stack, the number of times indicated by the counter.

Animation

Animation is most easily achieved by switching alternately between two screens; one on display, the other being updated in the background. Initially write two identical screens and display one while modifying the other, swap the screens over and display the modified screen while updating the one previously on display. The technique will produce a very stable display with quite slow switching rates.

Sound

SOUND CONCEPT OVERVIEW

Sound is generated via a Yamaha YM2149 programmable sound generator. The PSG contains three tone generators that produce the basic square wave tone frequencies for the A, B and C channels and a noise generator, that produces a frequency modulated pseudo random pulse width square wave, which may be combined with the tone generator outputs using the channel mixer. The output level can be fixed via the channel amplitude control using one of the three sixteen level D/A converters or varied by using the output of the envelope generator, which may be used to amplitude modulate the output of each mixer.

Sound control registers

The frequency of each tone generator (30Hz to 125KHz) is obtained by counting-down the 12-bit value of the tone registers (the coarse register sets the upper 4 bits and the fine register sets the lower 8 bits, range 001H to FFFH (1 to 4095). The standard PSG format is to produce a lower note for a higher count whenever a register count-down is performed.

The noise generator frequency is controlled by a 5-bit noise period register, value 01H to 1FH (1 to 31), producing a frequency range of 4Khz to 125Khz.

The mixer control register is a multi-function register that mixes the noise channels (defined by bits 3 to 5) and the tone channels (defined by bits 0 to 2) in all possible combinations to the input/output ports (bit 6 I/O, bit 7 port A or B).

The amplitude of a channel is controlled to one of sixteen fixed levels by the channel D/A converter register (lower 4 bits of the register) and only by setting the register to zero can the channel be turned off. The fifth bit of the amplitude control register is set to select the variable level output defined by the envelope generator.

The envelope generator comprises three registers, two provide the frequency variation and the third the format of the envelope. The frequency is determined by counting down the 16-bit value of the coarse and fine envelope registers range 0001H to FFFFH (1 to 65535). The shape and cyclic pattern of the envelope is defined by the lower 4 bits of the shape register (the amplitude register setting the level), the four bits provide for combinations of hold/cycle, reverse cycle on/off, ramp up/down and cycle hold pattern/reset to zero.

Parallel data I/O

The I/O register in the PSG is not associated with sound production, it provides a register to transfer 8-bit parallel data to and from the CPU bus to the I/O port A, there is no affect on any of the PSG's other functions.

Data is written to a peripheral device from the bus using the following steps:

Select enable register (mixer register)

Set bit 6 to '1' (set I/O port A to output)

Select I/O port A data store (I/O port A register)

Write data to PSG (write data to I/O port A register)

Once data has been loaded into the register, the data remains until further data is loaded, the system is reset, or

the register is switched to input mode.

Data is read from a peripheral device to the bus with the following steps:

Select enable register (mixer register)

Set bit 6 to '0' (set I/O port A to input)

Select I/O port A data store (I/O port A register)

Read data from PSG (read data in I/O port A register)

The register follows signals applied to the port, only by reading will the data be transferred to the bus.

SOUND CONFIGURATION REGISTERS

Access to the PSG should be in supervisor mode as the SR register is modified. The PSG registers are located for write at address($FF8800 – 16746596) as follows:

Offset Hex	Dec		
0	$0	Channel A fine tune	(8 bit)
1	$1	Channel A coarse tune	(4 bit)
2	$2	Channel B fine tune	(8 bit)
3	$3	Channel B coarse tune	(4 bit)
4	$4	Channel C fine tune	(8 bit)
5	$5	Channel C coarse tune	(4 bit)
6	$6	Noise generator control	(5 bit)
7	$7	Mixer control, I/O enable	(8 bit)
8	$8	Channel A amplitude	(5 bit)
9	$9	Channel B amplitude	(5 bit)
10	$A	Channel C amplitude	(5 bit)
11	$B	Envelope period fine tune	(8 bit)
12	$C	Envelope period coarse tune	(8 bit)
13	$D	Envelope shape	(4 bit)
14	$E	I/O port A	

Tone frequency calculations (registers 0 to 5)

The tone frequency is in the range 30.5Hz to 125Khz and may be calculated from the formula:

$$F = \frac{2 \times 10^6}{16 \times (256 \times CT + FT)}$$

where CT = coarse tone period
FT = fine tone period

Noise frequency calculations (register 6)

The noise frequency is in the range 4Khz to 125Khz and may be calculated from the formula:

$$F = \frac{2 \times 10^6}{16 \times Np}$$

where Np = noise period

The mixer control-I/O enable (register 7) bit functions take the following format:

0 1 2	3 4 5	6 7
Tone channels	*Noise channels*	*I/O port*
A B C	A B C	A B
If the bit is zero the channel is on.		If bit 0 port i/p

The channel amplitude (registers 8-10) bits have the following function:

-	-	-	M	x	x	x	x

M = 0: Fixed amplitude level 0–low to 15–high (xxxx)

M = 1: Amplitude determined by envelope shape

Envelope shapes

The envelope period (registers 11 & 12) of the shape is based on the 16-bit register value:

$$F_e = \frac{f_{clock}}{256 \times Ep}$$

where Ep = envelope period
f_{clock} = input clock frequency

The envelope shape/cycle control (register 13) bit settings produce the following range of sound envelopes:

Bits 0 1 2 3	*Function*	*Bits* 0 1 2 3	*Function*
x x 0 0		1 1 0 1	
x x 1 0		0 0 1 1	
0 0 0 1		1 0 1 1	
1 0 0 1		0 1 1 1	
0 1 0 1		1 1 1 1	

Bit 0 = Hold/cycle
Bit 1 = Reverse on/off
Bit 2 = Ramp up/down
Bit 3 = Cycle hold/reset zero

1 = bit set
0 = bit clear
x = don't care

GEM disk operating system overview

For those systems supplied with the operating system on disk; the system disk contains on the first two tracks, a cold start loader that loads the operating system image file (TOS.IMG) into high memory and then block loads it down into RAM memory at address $5000.

The TOS image file contains both the GEM and Atari ST extended operating systems, including:

CCP Console command processor: User interface to parse command line

BDOS Basic disk operating system: Access functions to the file system

BIOS Basic I/O system: Functions that interface peripheral device drivers

The operating system is always in memory above $400 and all modules reside permanently in memory, even those of disk based systems (unless the power is removed). After TOS is loaded, the remaining contiguous address space is called the transient program area (TPA) where TOS loads executable (command) files. The command files (programs) should not access absolute addresses or default TOS variables but use the BIOS and BDOS function calls.

Each transient program loaded into memory consists of the program segments (text, data and bss), a user stack and a base page. The 256 byte base page contains the direct memory address (DMA) buffer, at base page offset $80; the buffer contains the command tail, typically the input typed to an application installed as a TOS Takes Parameters program. Before the loaded program takes control, the address of the transient programs base page and a return address are pushed onto the user stack, 4(A7) and (A7) respectively.

Although the CCP can only load one program; the transient program itself can load further programs using GEMDOS function $4B, but must specifically supply the base

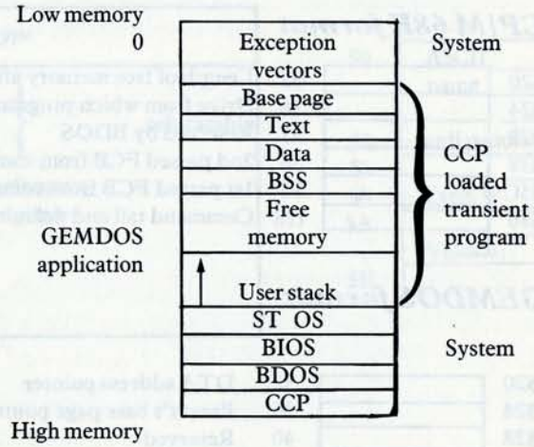
A return from a transient program may be achieved by:

An RTS as the last statement, returning via the return pushed onto the stack by the load function.

Executing a warm boot by calling extended BDOS function 0.

Typing CTRL c from the console during the execution of console output, printing a string or reading from the console buffer (functions 2, 9 and 10).

GEMDOS Memory model



Command file

The format of a command file is that of a header, two program segments (text and initialized data segments) and optionally a symbol table and relocation information. After the program is linked and loaded into memory, it contains additionally a zeroed uninitialized data (BSS) program segment and starts execution at the beginning of the text segment.

Not all assemblers provide for an uninitialised data section within the source code, this results in an executable program file on disk that is much larger than need be.

The operating system holds information on the data segments in a descriptor block (256 byte base page data structure) at the bottom of the TPA. The base page does not reside at a fixed address, its position is determined when it is created by the load a process function (GEMDOS function #$4B) and held in register D0.L.

The base page contents are initialized by the GEMDOS load function:

Base page format initialized by GEMDOS

$00		0	Base address of TPA
$04		4	End address of TPA + 1
$08		8	Base address of text (code)
$0C		12	Length of text (code)
$10		16	Base address of initialized data
$14		20	Length of data
$18		24	Base address of BSS uninitialized data
$1C		28	Length of BSS uninitialized data

There are slight differences between small sections of the original CP/M 68K and GEMDOS base page formats as follows:

CP/M 68K format

$20	32	Length of free memory after BSS	
$24	36	Drive from which program loaded	
$25	37	Reserved by BDOS	
$38	56	2nd parsed FCB from command line	Set by CCP
$5C	92	1st parsed FCB from command line	Set by CCP
$80	128	Command tail and default DMA buffer	Set by CCP

GEMDOS format

$20	32	DTA address pointer
$24	36	Parent's base page pointer
$28	40	Reserved
$2C	44	Pointer to environmental string
$80	128	Command line image (typically the entry to a dialog box for a TTP application)

File header format

GEMDOS file header and program segments take the format:

File header

$00	0	Data and BSS contiguous 601AH else 601BH
$02	2	Number of bytes in text segment
$06	6	Number of bytes in data segment
$0A	10	Number of bytes in BSS
$0E	14	Number of bytes in symbol table
$12	18	Reserved (zero)
$16	22	Start of text segment and prog execution
$1A	26	Zero if no relocation bits

If data and BSS are not contiguous (first field equals 601BH)

$1C	28	Start address of data segment
$20	32	Start address of BSS

Note: 601AH is a `BRA.S` instruction that bypasses the file header data segment.

The Atari OS does not support segmented files.

The symbol table consists of fourteen bytes that specify a null padded 8-character name, the type of symbol and the symbol value (address etc).

Hex	*Symbol type*	
100	BSS	relocatable
200	Text	relocatable
400	Data	relocatable
800	External reference	
1000	Equated register	
2000	Global	
4000	Equated	
8000	Defined	

$0	ASCII name
$7	null padded
$8	
$9	Type.W
$A	
$E	Value.L

Relocation table

The linker optionally produces a relocatable executable file and places the relocation information in the GEM file header.

Type	Sort of address referenced (File header offset $1a)
00	No relocation data required, absolute references
01	References relative to data segment base address
02	References relative to text segment base address
03	References relative to bss base address
04	References an undefined symbol
05	References the upper word of a longword, the next relocation word determines 'absolute' or 'relative'
06	16-bit PC-relative reference
07	First word of an instruction not to be relocated

Relocation word format

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Relocation word (bits 0–2)

Index value to a symbol in the symbol table (starting at zero) (bits 3–15)

If the offset byte is 1, then a multiple byte offset based on the following table is used to determine the actual offset:

Offset byte value	*Relocation data*
$00	End of relocation data
$01	Add 254 from current location and decode next byte
$02 . . $FF	Add byte value from current location

When the program is loaded into memory at a location other than where it was linked, BDOS computes an offset and adds the offset to the address of the relocation words in the text and or data segments.

GEMDOS function $4B (75 decimal) loads or executes a program.

ATARI ST FILE SYSTEM

GEM contains a fairly comprehensive set of file manipulation facilities, they enable the programmer to write software that provides multiple access file sharing and file protection, periodic file updates and selective backups. The file facilities are:

Code			
Dec	*Hex*	GEMDOS function	Comments
60	3C	Create file	
61	3D	Open file	Invoked by filling a parameter block with the number of the function, the parameters and any other relevant data.
62	3E	Close file	
63	3F	Read a file	
64	40	Write a file	
65	41	Delete file	Returns are in D0.L. A zero indicates ok.
66	42	Seek file pointer	
67	43	Get/set attributes	
69	45	Duplicate file handle	Where data is returned, D0 contains the address of the data return block.
70	46	Force file handle	
78	4E	Search for first	GEM uses the stack as the parameter block.
79	4F	Search for next	
96	56	Rename file	
97	57	Get/set date/time stamp	

ATARI ST DISK SYSTEM

The Atari ST 3.5" disk uses soft sectored disks of the following format:

Bytes/sector	512	
Sectors/track	9	
Tracks/side	80	
Sides/media	1	2
Unformatted	360K	720K

The GEM BIOS interfaces (Basic input/output systems) make the hardware dependent interface to the floppy disk drives. These communicate with the drives as follows:

Select the drive, the side, the track and then the number of sectors from the track that will be read to a buffer or written from the buffer to the disk.

GEM BDOS is fairly basic in terms of disk operations but has extensions to handle tree type directories.

Code		*GEMDOS function*
Dec	*Hex*	
14	0E	Set default drive
25	19	Get default drive
54	36	Get drive free space
57	39	Create a subdirectory
58	3A	Delete a subdirectory
59	3B	Set current directory
71	47	Get current directory

A file does not use consecutive disk sectors as there is insufficient time to identify, read and or write a record via software, and to locate a specific track via hardware. The record spacing (skew) is usually 6 sectors between adjacent file segments.

ATARI ST BIOS COMPARISONS

The ST contains device dependant input/output utilities that handle the interface between the device independant routines and the hardware, the ST BIOS and GEM BIOS utilities are supplemented by the A-line primitives which provide rapid screen control.

The GEM type BIOS handles the input/output to the peripheral devices: parallel port, RS232 port, console, midi interface and intelligent keyboard. There is also a basic disk read/write to sector and a facility to check that the disk has not been removed or replaced.

The ST extended BIOS also controls the input/output to the midi interface, intelligent keyboard, console and disk read/write, but additionally includes the control of a mouse, joysticks, sound and of the screen colours; the disk facility is augmented by format.

The A-line routines are the VDI graphic primitives which are not program transportable and therefore included here, they enable control of the mouse and pixel–line–sprite–screen graphics.

Interrupt handler overview

The operating system provides the machine code programmer with access to the interrupt handler.

Every fiftieth of a second control is transferred from the operating system to a routine at the address designated in the system variables at $68 (104 decimal), the system interrupt handler (vertical blank interrupts). The handler provides a timing facility, sets the screen parameters and current device driver installation and entry points.

System Initialization

The ST in general follows a predefined initialization sequence on power-up, with slight variations for the different operating systems.

```
System reset
                        The supervisor stack pointer (SSP) &
ssp --> $60xxxxxx       program counter (PC) are set from $0
pc --> $00FC0020        and $4 respectively, the SSP is
                        garbage until system is sized. The
move.w #$2700,SR        Interrupt priority level (IPL) is
reset                   set to seven and a hardware reset
cmpi.l #$FA52235F,      executed. Checks for a diagnostic
$FA0000                 cartridge, if present causes
bne cmpi_nxt            a return address to be set in A6 and
lea $8(PC),A6           execution of the diagnostic routines
jmp $FA0004             commenced.
.
.
.
cmpi.l #$31415926,      A check is made to see if memory has
$426                    previously been sized. If not,
bne psgset              jump past memory sizing routine. If
 move.l $42A,D0         this is a soft reset, the bailout
tst.b $42A              vector may be valid. First check if
bne psgset              the MSB is zero, secondly that the
btst #$0,D0             vector is to an even address, if not,
bne psgset              jump past the reset handler.
movea.l D0,A0           Set A0 to point to the reset handler,
lea $4(PC),A6           set A6 to the return address and
jmp (A0)                jump to reset handler.
lea $FFFF8800,A0        Set A0 to PSG configuration register
move.b #$7,(A0)         base and set porta & portb to output,
move.b #$C0,$2(A0)      activate general purpose output and
move.b #$E,(A0)         through output porta deselect the
move.b #$7,$2(A0)       disks.
move.b #$0,$FFFF820A    Set sync mode to external 50Hz and
lea $FFFF8240,A1        A1 to base of pallette table.
move.w #$F,D0           Set up a count in D0 to shift the
lea $28A(PC),A0         default hardware pallette colors to
move.w (A0)+,(A1)+      A0 which points to the default color
dbf D0,loop             table.
.                       Size both memory banks and perform a
move.l #$752019F3,$420  memory test. Set 'memory sized' and
move.l #$237698AA,$43A  'memory tested' flags in the system
.                       variables table. Set up screen,
move.l #$8900,$432(A5)  vblank queue entries, BIOS entry
```



```
lea $D50,A7             point and supervisor stack.
.                       Run type '0' cartridge applications.
.                       Point A3 and A4 at RTE and RTS
.                       resply Test diagnostic cartridge.
.                       Initialise exception vectors to
.                       terminate process handler except for
move.l rte,$14          divide by zero which is RTE'd.
move.l #$FC0324,$70(A5) Set vblank handler entry address.
move.l rte,$68(A5)      Kill hblank handler entry address.
move.l rte,$88(A5)      Initially empty trap#2 handler.
move.l #$FC03C0,$B4(A5) Set up trap#13 handler address.
move.l #$FC03BA,$B8(A5) Set up trap#14 handler address.
move.l rts, $400(A5)    Default timer tick vector to RTS.
move.l #$FC03B6,$404(A5)
                        Set up the critical error handler
move.l rts,$408(A5)     and default the terminate vector.
move.l #$550,$4A2(A5)   Set up BIOS register save area
suba.l A5,A5            pointer Zero page pointer.
.
suba.l A5,A5            Intialize vblank vector list.
move.w $454(A5),D0      8 nvbls into D0
lsl.w #2,D0             multiply by four to
move.w D0,D1            create queue length in D1.
bsr make_spc            Routine to create a space of
and.l #$FFFF,D0 8       longwords in high memory.
btst #$0,D0             make
beq jump1                   address
addq.w #$1,D0                   even
movea.l $436,A0         current memory top
suba.l D0,A0            'come on down'
move.l A0,$436          reset memory top
move.l A0,D0            and put in D0
rts
move.l A0,$456(A5)      vblqueue start address
subq #$1,D1             zero
clr.b (A0)+                 the
dbf D1,clr_byte                 queue

                        Initialize screen resolution.
```

```
move.w #$F8FF,SR      Enable all interrupts except Hblank
                      by setting IPL to 3.
                      Run type '1' cartridge applications
                      Initialize GEMDOS - set up a DOS
                      disk buffer chain & memory manager.
                      Run type '3' cartridge applications
                      Attempt to boot from floppy and
                      execute if successful, if not poll
                      devices on DMA bus for logical boot
                      sector zero, execute if successful.
                      Any 'returns' continue polling the
                      devices in sequence.
                      Turn on the cursor.
                      Execute file COMMAND.PRG?
                      otherwise construct a default
                      enviroment and execute AES.
                      Initiate a RESET on a return.
```


CARTRIDGE SOFTWARE

There are two types of cartridge which may be plugged into the ST; diagnostic and program cartridges. The cartridge program header format is as follows:

Base address
$FA0000 (16384000)

$0	C-FLAG	0	Only the first header contains a flag which denotes the presence of a cartridge. flag: #$FA52255F = diagnostics #$ABCDEF42 = program/data
$4	C-NEXT	4	Pointer to next application header, a null indicates no additional applications.
$8	C-INIT	8	Pointer to application initialization code, if zero there is no initialization code. The longword high byte is unused in the 24 bit address and starts applications as follows: bit 0–set run before interrupt vectors and memory initialized. bit 1–set run before GEMDOS initialized. bit 2 unused bit 3–set run before disk boot. bit 4 unused bit 5–set Application is a desk accessory bit 6–set Application not a GEM application (no AES calls) bit 7–set Requires command line parameters before execution.
$C	C-RUN	12	Pointer to application entry point.
$10	C-TIME	16	Time } DOS format time and date stamps.
$12	C-DATE	18	Date } DOS format time and date stamps.
$14	C-BSIZ	20	The size of the applications BSS segment allocation. The OS must allocate the BSS before invoking any run code. Set to zero if not applicable.
$18	C-NAME	24	The ASCII name (max 12 chars) terminated with a zero (NNNNNNNN.EEE).

Diagnostic cartridge: The ST hardware will not be initialized and a return address is held in A6, the stack pointer is trashed. The cartridge software is responsible for sizing memory and setting the hardware registers as required.

Cartridge software: Application headers are strung together in a linked list, so there may be any number of applications on one cartridge.

BOOT SECTORS

To write software that will auto run from disk, the programmer must produce a boot sector that contains a loader program which transfers the program from disk to memory before bringing up GEM.

The boot sector follows IBM PC format and contains:

The volume serial number
  24 bit number generated when the media is formatted

BIOS parameter block (BPB)
  Sector size in bytes
  Number of sectors/cluster
  Cluster size in bytes
  Length of root directory in sectors
  Size of a File Allocation Table (FAT – in sectors)
  Sector# of start of second FAT
  Sector# of first data sector
  Number of data clusters on disk
  Flags

Optional boot code and boot parameters

During initialization the boot sector is loaded into a buffer and the executable boot sector code tested for a word checksum of #$1234. If satisfactory a subroutine jump is made to the beginning of the position-independent code in the buffer.

When a `get BIOS parameter block` call is made, the BIOS reads the boot sector (normally created when the volume is formatted), and returns an error indication if any critical parameter fields are zero.

The 24-bit volume serial number, written when the media is formatted, is used to determine whether or not a disk has been changed.

The `protobt` extended BIOS call (dec 18) is used to create the boot sector (see chapter 3), which is written to track 0, side 0, sector 1.

BIOS boot parameter block (normally written when the volume is formatted)

Note: word storage is low byte at the low address (even) as per 8086 and not the usual 68000 mode.

The BIOS parameter block is compatible with MS-DOS version BPB, but will only read and write sectors written by another WD1772A disk controller.

$0	BRA.S	0	Branch to boot code
$2	oems–space	2	Space reserved for OEMs use
$8	Volume serial# #$000000	8	24-bit volume serial number (used to determine disk changes)
$B	BPS $00 #$02	11	Number of bytes/sector
$D	SPC #$02	13	Number of sectors/cluster
$E	RES #$01 #$00	14	Number of reserved sectors (at start of media incl. boot)
$10	NFATS #$02	16	Number of file allocation tables on media.
$11	NDIRS #$70 #$00	17	Number of directory entries
$13	NSECTS #$D0 #$02	19	Number of sectors on media (including reserved)
$15	MEDIA #$F8	21	Media descriptor - not used by ST
$16	SPF #$05 #$00	22	Number of sectors/FAT
$18	SPT #$09 #$00	24	Number of sectors/track
$1A	NSIDES #$01 #$00	26	Number of sides on media
$1C	NHID #$00 #$00	28	Number of hidden sectors-not used
$1E	boot code	30	Start of code, if any
$1FE	The last word	510	Used for checksum
$200		512	

BOOT LOADER

The boot loader resides in the boot sector and is used during system initialization to load an image file or a contiguous set of sectors; it is also used to load GEM from disk on early ST models. The format of the loader is:

$0	boot sector	0	The standard BIOS parameter block
$1E	execflg	30	The word copied to `cmdload` flag
$20	ldmode	32	If `lmode` = 0 load file, if not 0 load `sectcnt` sectors beginning at `ssect`
$22	ssect	34	If `lmode` <> 0 load from here
$24	sectcnt	36	If `lmode` <> 0 load `sectcnt` sectors
$26	ldaddr	38	Load address of file or sectors
$2A	fatbuf	42	Address for FAT and DIR sectors
$2E	fname	46	Filename: `8 character name, 3 character extension` (valid if `ldmode` is zero)
$39	(reserved)	57	Reserved
$3A	boot code	58	The executable code

Some software tools require the six bytes reserved for OEMs at offset $2 to contain the ASCII text 'loader'.

The loader can load any file from disk regardless of where it appears in the directory or whether it has the form of contiguous sectors or not.

An image file contains no header or relocation information and is an exact copy of the program to be executed.

BOOT ROM

The initialization of the system from the boot ROM follows the predefined pattern of a RESET with some system variables installed and pretty color screen graphics to keep the operator from getting bored.

The boot `dir` and 2nd FAT buffer are read into memory starting at `membot`. `TOS.IMG` is loaded starting at $40000 and an error code produced if the file is not found. The memory $10000 to $20000 is used for screen buffers and should not be used initially for any code or data.

The first ST's sold contained a small 32K boot ROM that loaded the operating system from disk. The boot ROM contains a small sub-set of the BIOS, just sufficient to read an 80 track, BPB floppy disk boot sector from either drive into memory and then execute it.

Trap 13 – GEM BIOS functions implemented

Code	*Name*	*Function*
4	`rwabs`	Read/write sectors (read only)
7	`getbpb`	Get BIOS parameter block

Trap 14 – extended functions implemented

Code	*Name*	*Function*
1	`ssbrk`	Reserve x bytes from top of memory
8	`floprd`	Read sectors from floppy disk

All other BIOS facilities are not loaded into the system until a later stage. The first 100 bytes of disk TOS relocate TOS.IMG at $5000 from where it takes control.

The first TOS implementation uses the following disk parameters:

80 track, single sided BIOS parameter block

Bytes/sector	512	# sides/media	1
Sectors/cluster	2	# hidden sectors	0
Reserved sectors	1	Load address	$40000
# of FATs	2	FAT/directory buffer	$8000
# of root dir entries	7	Volume serial number	0
# of sectors on media	720	Media descriptor byte	F8H
# sectors/FAT	5	Filename	`TOS.IMG`
# sectors/track	9		

Atari ST peripheral device communications

COMMUNICATIONS OVERVIEW

The ST supports serial and parallel communications through dedicated RS232 and parallel ports, and permits two further communicaton channels to be opened through the MIDI and DMA ports.

The serial RS232 communication port accomodates hardware data control based on the PSG I/O port A, RTS and DTR outputs, and the MFP MK68901, CTS, DCD and RI inputs, and Xon/Xoff software data protocol at transmit and receive baud rates in the range 50 to 19200 baud. The port is generally used to interface with a printer, modem or another computer. The MFP is located at $FFFA00 (16775680) and the PSG at $FF8814 (16746610).

The general purpose parallel port interface provides bi-directional 8 bit communications for printer operation. The port is based on the MFP MK68901 (busy control), the PSG I/O port A bit 5 (strobe control) and the PSG I/O port B (data transfer). The control is limited to a busy signal, acknowledge is not supported and data transfer is at a typical rate of 4000 bytes/s.

The MIDI interface provides an asynchronous, current loop, serial data (one start bit, eight data bits and one stop bit) communications channel at 31.25 Kbaud. The MC6850 port controller may be reconfigured for most forms of RS232 interface via the control/status register situated at address $FFFC04 (16776196).

The intelligent keyboard interface is also controlled by an MC6850 ACIA, but there is no external access provided to the port, which is of limited use other than accessing the ikbd command set; for reading and or writing to the clock, joysticks, mouse and perhaps reconfiguring the keyboard.

The floppy disk interface is based on the Western Digital WD1772A disk controller and is limited to supporting two drives.

The DMA interface is provided by an ULA device, access is through the control/status configuration registers at $FF8600 (16746084) et seq.

RS232 INTERFACE

General

Data is transmitted and received via an RS232 interface as a sequence of ones and zeroes (bits) along a three wire link, one wire being ground, one for transmitted data and the other for the received data. Information is sent as 'characters' and each character is prefixed by a start bit (a one) and terminated with either one or two stop bits (zeroes). Providing the sending and receiving devices are set to the same speed (baud), then the stop and start bits act as a timing signal to each 'character' sent. Occassionally error detection is incorporated in the form of a parity bit. If the count of ones in the character is even, then the eighth bit is a warning of data errors in the transmission. Providing the transmitting and receiving station agree on the protocol used, then communications will be reasonably straightforward!

The port is reconfigured using the sequence:

a: Save current MK68901 register contents

b: Disable Rx and Tx enable bits

c: Set flow control mode

d: Set baud rate

e: Set RS232 registers

f: Re-enable Rx and Tx enable bits

The extended BIOS call #$0F (15) enables selective reconfiguration of the RS232 port according to a block of parameters pushed onto the stack:

```
move.w sync_char,-(SP)  * Pushing (see chapter 3)
move.w tx_status,-(SP)  * -1
move.w rx_status,-(SP)  * leaves
move.w usrt_cntl,-(SP)  * parameter
move.w flow_cntl,-(SP)  * unchanged

move.w baud_rate,-(SP)  * Set timer D
move.w    #15,-(SP)     * push RS232 config
trap      #14           * call function
add.w     #14,SP        * tidy stack, jump 7 words
tst.w     D0            * test for error
rts                     *
```

Data is passed through the interface using extended BDOS calls to the auxiliary device, the RS232 port.

PARALLEL PORT INTERFACE

General

Data is transmitted and received via a parallel port interface in blocks of 8 (sometimes 7) data bits, set either as ones or zeroes to form a character byte. The character is 'framed' by a strobe signal enabling the receiving device to read the character transmitted, which may be printed immediately or saved in a buffer for subsequent printing. At some stage the printer will not be able to accept further input and will send a busy signal to stop the transmitter from sending additional data. The acknowledge signal is sometimes used to indicate that the printer is no longer busy, occasionally this signal line is omitted and the busy line also provides the 'not busy' signal.

Data is passed to and from the interface using the following procedures:

Write data:

a: Check the busy line for high
If line low, monitor until high
or time out set CPU D0 register to 0

When high

b: Set PSG I/O B port to output, use IPL 7
c: Place data into the PSG's B output register
d: Switch strobe line on
e: Switch strobe line off, set CPU D0 register to –1

Read data

a: Set PSG I/O B port to input
b: Switch strobe line off
c: Check busy line for high
loop till high
d: Switch strobe on
e: Get data from PSG's B output register

As the status register is affected, the above procedures should be performed in supervisor mode.

MIDI INTERFACE

General

The MIDI (musical instument digital interface) sequential circuits provide for integrated operation of music synthesizers, sequencers, drum boxes etc. which have the MIDI interface. The ST operates as a data store for a large number of notes/voices which may be sent to different instruments (channels), and played together in sequence and time as music. The data may be 'recorded' from a tune previously played, edited and/or synthesized by entering new data in step–time–note format into the store for later retrieval.

The MIDI bus provides 16 channels in one of three networking modes. OMNI, the default where all units are addressed together and transmit and receive on all channels. POLY where all the units are individually addressed and receive on one channel only, data assigned to non-existent channels is ignored. MONO where the voice of each unit is addressed seperately, providing different channels for individual voices within one synthesizer.

The information transmitted is priortised and sent as bytes, the most significant bit signifying either status (1) or data (0). The priority order is:

System reset	Set defaults
System exclusive	Manufacturers unique data
Sequential circuits	Roland, Yamaha etc.
System real time	Synchronization
System common	Broadcast
Channel	Note selection, program data etc.

The MIDI port supports the optional through port which merely provides the MIDI in signals at the MIDI out port.

The MIDI interface operates in RS232 current loop mode at 31.25 Kbaud. It may be reconfigured by resetting the control/status registers.

The Atari ST's extended BIOS enables the programmer to reconfigure the MIDI port.

MIDI control/status register functions (write only)

Control/status register located at address $FFFC04 16776196
Data register offset $2

Control register functions (write only)

Bit	*Divide select*	*Bits*	*Data format*			*Bit*	*RTS format*	*Bit*	Interrupt
0 1		2 3 4	#Bit	Parity	Stop	5 6	Tx on/off	7	enable
0 0	by 1	0 0 0	7	even	2	0 0	off RTS=		Interrupts
0 1	by 16	0 0 1	7	odd	2	0 1	on low		enabled by
1 0	by 64	0 1 0	7	even	1	1 0	off RTS =		bit 7 = 1:
		0 1 1	7	odd	1		high		Rx data
1 1	Master	1 0 0	8	—	2	1 1	off RTS =		reg full,
	reset	1 0 1	8	—	1		low		Overrun,
		1 1 0	8	even	1	Tx break level			DCD low to
		1 1 1	8	odd	1	on Tx data o/p			high step

Status register functions (read only)

Bit	*Name and Function*
0	Rx data register full Received data in register ready for CPU read
1	Tx data register empty Transmitted data sent, load with next character to transmit
2	Data Carrier Detect Indicates modem state – Carrier present
3	Clear to Send Indicates modem state – Master reset – no change
4	Frame error Character synchronization error
5	Rx over-run Characters have been lost from stream
6	Parity error Only active if parity selected
7	Interrupt request Read received data register or write to transmit data register

INTELLIGENT KEYBOARD INTERFACE (IKBD)

The intelligent keyboard functions through a MC6850 ACIA device whose control/status register is located at address $FFFC00 (16776192), and functions like the MIDI interface. There is no external access to this port so there is little point in reconfiguring, but it can be used to transmit and receive data or commands from the keyboard, mouse, joystick and clock using the following facilities:

Keyboard

Return keycodes

Mouse

Set mouse button action (keys, on press/on release)
Set mouse position relative (default)
Set threshold level per 'click'
Set mouse position absolute
Set scale ('clicks' per movement)
Read/write mouse position
Set mouse to simulate cursor motion codes
Set Y origin top/bottom
Disable/pause/resume mouse operation

Joystick

Enable joystick (default)
Disable, act on request only
Interrogate joystick
Set monitoring (serial line, joystick and clock)
(serial line, button 1 and clock)
Set keycode mode (variable 'click' rate)
Disable joystick

Clock

Set date and time
Read date and time

Program control

Load data into ikbd memory
Read data from ikbd memory
Execute intelligent keyboard (ikbd) program

A status inquiry command returns a null padded 8-byte packet detailing the current mode and parameters of a specific function, the packet may be stored and later used to restore the status of the keyboard by modifying the header byte and returning the data as a command.

The keyboard scancodes do not maintain complete compatibility with IBM PC key scancodes. Appendix D provides the major differences due to the non-availability of certain keys on the ST keyboard. The additional ST keys are mapped into unused CTRL_ and ALT_ function scancodes.

To detect CTRL_ and ALT_ function key combinations, execute a BDOS or BIOS `getchar` call followed by a BIOS `kbshft` call (#$0B).

FLOPPY DISK INTERFACE

The floppy disk interface is based on an on-board Western Digital WD1772A disk controller and that can support a maximum of two drives.

The floppy disk read/write sequence of events is:

a: select floppy drive 0 or 1 (PSG I/O port A)

b: select floppy side 0 or 1 (PSG I/O port A)

c: load DMA base address and counter register

d: toggle read/write to clear status (DMA mode cntrl register)

e: select DMA read or write (DMA mode control register)

f: select DMA sector count register (DMA mode cntrol register)

g: load DMA sector count register (DMA mode trigger)

h: select FDC internal command reg (DMA mode control register)

i: issue FDC read or write command (Disk controller register)

j: DMA active until sector count zero (DMA status register) do *not* poll during DMA active.

k: issue FDC force interrupt command on multi-sector transfers except at track boundaries (Disk control register)

l: check DMA error status, non destructive (DMA status register)

The DMA configuration registers are at the base address $FF8600 (16746084) and the following offsets:

4	$4	Disk controller data access	
6	$6	DMA read – mode control, write FIFO	
9	$9	DMA base high	*set last*
11	$B	DMA base medium	
13	$D	DMA base low	*set first*

The PSG configuration registers are at base address $FF8800 (16746596) and the following offset:

2	$2	PSG write port Bit 0 floppy side Bit 1 floppy drive 0 Bit 2 floppy drive 1

There is no hardware support for sensing disk removal, therefore this facility must be performed in software.

Formatting a floppy disk

The following procedure illustrates the technique used in formatting a floppy disk:

Extended BIOS calls

The standard ST format is:

flopfmt #10

- Sides: 1 or 2
- Sectors per track: 9
- Tracks: 80
  - No interleave and the first two tracks zeroed (to 0 FAT and directory sectors, either sector bad and the media is unusable).

protobt #18

- Use disk type parameter 2 or 3
- Serial # param, random or #$1000000
- Execute flag usually zero, non zero if it contains loader code etc. that is to execute when the disk is booted.

flopwr #9

- Write boot sector (prototyped in buffer to track 0, side 0, sector 1 of the disk.
- Do not use rwabs call.

The WD1772 'write track' codes used to format a track are:

Double density format: issue a 'write track' command and load the following values into the data register. There is a data request for every byte written.

			ID field							
# Bytes data	60 #$4E	12 #$00	3 #$F5	ID #$FE	Trck # 0-$4F	Side # 0-1	Sect # 1-9	Len #	CRC 1	CRC 2
# Bytes data	22 #$4E	12 #$00	3 #$F5	ID #$FB	256 data	CRC 1	CRC 2	40 #$4E		1401 #$4E End of track
			Data field							

Length = #512 bytes/sector (usually 2)

WD1772 DMA channel interface

The WD1772 is interfaced through the DMA channel via the following procedure:

To initialize the WD1772:

```
move.w #$190,$FF8606      ;Clear the fifo by toggling r/w
move.w #$90,$FF8606       ;and leave in the write state.
move.b #xx,dmalow         ;Set up dma address pointer in
move.b #xx,dmamid         ;low to high order
move.b #xx,dmahigh        ;$FF860D, $FF860B & $FF8609
                          ;respectively
```

The following addresses are used by the WD1772

$80	128	command/status register
$82	130	track register
$84	132	sector register
$86	134	data register

To address the WD1772:

```
move.w #$yy,$FF8606       ;The FDC rquires two writes to
move.w #$zz,D7            ;access the registers, the first
delay                     ;write selects the FDC register
move.w D7,$FF8604         ;and the second write modifies
delay                     ;the register
rts
```

To transfer from memory to floppy the values must be ORed with #$100 and #$FF written to $43E to prevent TOS from changing the value in address $FF8606. When the operation is complete the byte in $43E, the floppy lock variable, must immediately be zeroed.

To seek to a track:

```
move.w #$86,$FF8606       ;Select the data register
move.w #$4F,$FF8604       ;Write seek track ($4F last track)
move.w #$80,$FF8606       ;Select command register
delay                     ;Wait for drives
move.w #$17,$FF8604       ;Seek with verify.
```

The FDC will generate an interrupt when the seek is finished, it can be polled at $FFFA01 where bit 5 is zeroed. Errors are read from $FF8604, the read clearing the interrupt bit.

To transfer data:

```
move.b #$xx,dma          ;set up dma address - clear fifo
move.w #$190,$FF8606
move.w #$1,$FF8604       ;512 byte size limit of transfer
                         ;write sector# (1..9)

                         ;write track# (#$00.#$27) use #$A6
                         ;write track# (#$28.#$4F) use #$A4
                         ;read track#, use #$84
```

Do not use read/write multiple sector commands as they require a force interrupt command which is slower than re-executing a read or write.

To format a track:

```
write track# (#$00.#$27) ;use #$F6
write track# (#$28.#$4F) ;use #$F4
```

Write data to the drive beginning and ending with the index pulse.
It takes about #$1A00 bytes to fill a drive running at 3%.

The existing format command produces 9 sectors of 512 bytes per track.

Do not change the id-field, the fourth byte is used to count the number of bytes to transfer, and to locate the CRC data field. It may produce incompatibilities with TOS if changed. The side number can be read by making a read address command three times without clearing the dma fifo or changing the dma pointer.

To write an entire track:

The entire track can be written as one long sector and then read back, without any error checking, using the read track command if the following format is used:

Index pulse followed by:

#$00 a minimum of 12 bytes for lock-on
#$F5 3 bytes for synchronization

The #$F5s generate $A1s with a missing clock pulse to allow for alignment.

DMA INTERFACE

There is only one direct memory access (DMA) channel which is shared by both low and high speed 8-bit device controllers. The configuration registers hold the 3 byte base address of the DMA operation which is performed through a 32-bit FIFO programmed by the DMA mode control register.

The hard disk read/write sequence of events is:

a: load DMA base address

b: toggle read/write to clear status (DMA mode control register)

c: select DMA read or write (DMA mode control register)

d: select DMA sector count register (DMA mode control register)

e: load DMA sector count register (DMA mode trigger)

f: select HDC internal command reg (DMA mode control register)

g: issue HDC read or write command (Disk controller register).
1st command A0 set to 0, set to 1 for remaining commands.
Each byte command is acknowledged with an interrupt.
After last command byte set hard disk sector count bit 1.

h: DMA active until sector count zero (DMA status register) – do *not* poll during DMA active.

i: check DMA error status, non destructive (DMA status register)

j: check HDC status byte and if necessary perform an ECC correction following a verify track or read sector command.

The DMA configuration registers are at the base address $FF8600 (16746084) and the following offsets:

4 ($4)	Disk controller data access
6 ($6)	DMA `read_mode` control, write FIFO
9 ($9)	DMA base high
11 ($B)	DMA base medium
13 ($D)	DMA base low

The DMA registers are used to perform the floppy disk data transfers but may also be used for hard disk and other high speed data interfaces, bearing in mind the restriction of one DMA operation at a time.

The port is used for both high speed (upto 8 Mbit/s) and low speed (250 to 500 Kbit/s) DMA operations.

Any modification of the DMA base address or counter register requires that they be set in low–mid–high order.

DMA bus boot code

The following code, which is typical of the ST's BIOS, attempts to load boot sectors from devices on the DMA bus. The code shows typically how the DMA bus is used and provides the timeout and the command characteristics expected from bootable DMA bus devices.

```
gpip      equ  $FFFFFA01 * .B   68901 input register
dskctl    equ  $FFFF8604 * .W   controller data access
fifo      equ  $FFFF8606 * .W   DMA mode control
dmahigh   equ  $FFFF8609 * .B   DMA base high
dmamid    equ  $FFFF860B * .B   DMA base mid
dmalow    equ  $FFFF860D * .B   DMA base low
flock     equ  $43E      * .W   DMA chip lock variable
dskbuf    equ  $4C6      * .L   1K disk buffer
Hz_200    equ  $4BA      * .L   200 Hz counter
bootmg    equ  #$1234    * .W   boot checksum
```

Try to boot from DMA device

```
dmaboot   moveq  #0,D7        * # devices to try (eight)
dmb_1     bsr    dmaread      * try to read boot sector
          bne    dmb_2        * failed -- next device
          move.l dskbuf,A0    * disk buffer pointer in A0
          move.w #$00FF,D1    * checksum #$100 words
          moveq  #0,D0        * initialize checksum
dmb_3     add.w  (A0)+,D0     * add a word
          dbra   D1,dmb_3     * until #$100 counted
          cmp.w  #bootmg,D0   * Is it a boot sector
          bne    dmb_2        * No -- next device
          move.l dskbuf,A0    * disk buffer pointer in A0
          jsr    (A0)         * run the code.
dmb_2     add.b  #$20,D7      * next device
          bne    dmb_1        *
          rts
```

Try to read DMA bus device boot sector

```
dmaread   lea    fifo,A6      * DMA control register
          lea    dskctl,A5    * DMA data register
          st     flock        * DMA lock against vblank
          move.l dskbuf,-(SP) *
          move.b 3(SP),dmalow * set up DMA pointer
          move.b 2(SP),dmamid *
          move.b 1(SP),dmahigh *
          addq   #4,sp        *
          move.w #$098,(A6)   * toggle r/w, leave at read
          move.w #$198,(A6)   *
          move.w #$098,(A6)   *
```



```
          move.w #1,(A5)      * write sector count reg = 1
          move.w #$088,(A6)   * DMA bus select (not SCR ?)
          move.b D7,D0        * D0.l to device# + command
          or.b   #$08,D0      *
          swap   D0           * D0.l=xxxxxxxxDDD01000
          move.w #$088,D0     * xxxxxxxx010001010
          bsr    wcbyte       * write cmd and wait for IRQ
          bne    dmr_q        * error exit on timeout
          moveq  #3,D6        * write cmd  $00
          move.l #$00008A,D0  *      cntl $8A
dmr_1p    bsr    wcbyte       *      four times
          bne    dmr_q        * error exit on timeout
          dbra   D6,dmr_1p    *
          move.l #$00000A,(A5) * write final byte
          move.w #400,D1      * 2s timeout limit
          bsr    wwait        *
          bne    dmr_q        * error exit on timeout
          move.w #$08A,(A6)   * select status register
          move.w (A5),D0      * get DMA return code
          and.w  #$00FF,D0    * mask for error code only
          beq    dmr_r        * return if ok
dmr_q     moveq  #-1,D0       * set error return (-1)
dmr_r     move.w #$080,(A6)   * reset DMA chip for drivr
          tst.b  D0           * test for error return
          sf     flock        * unlock DMA chip
          rts                 *
```

Write ASCII command byte and wait for IRQ

```
wcbyte    move.l D0,(A5)      * write disk controller data
          moveq  #10,D1       * wait 0.05s
wwait     add.l  Hz_200,D1    * set D1 to timeout
ww_1      btst.b #5,gpip      * disk finished
          beq    ww_w         * ok return
          cmp.l  Hz_200,D1    * timeout yet?
          bne    ww_1         * no -- try again
          moveq  #-1,D1       * set error return (-1)
ww_w      rts
```

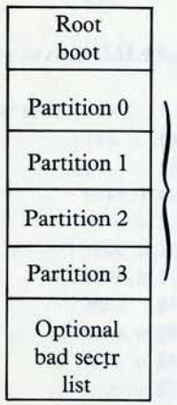
Hard disk partitioning

Logical sector #0 contains information on the four possible hard disk partitions:

	Offset					
hd_siz	$1C2		Total size of the disk in sectors			
p0_flg	$1C6		Non zero to show partition exists, bit 7 set for BIOS boot partition			
p0_id.	$1C7		3-byte field identifier. 'GEM' for GEMDOS			
p0_st	$1CA		Partition start logical sector number			
p0_siz	$1CE		Size of partition in logical sectors			
			Three further optional partitions			
px_flg	$1D2		$1DE		$1EA	
px_id	$1D3		$1DF		$1EB	
px_st	$1D6	2nd	$1E2	3rd		4th
px_siz	$1DA		$1E6		$1F2	
bsl_st	$1F6		Staring sector of the bad sector list			
bsl_cnt	$1FA		Number of bad sectors			
reserved	$200					

An ST disk may contain up to four partitions, the first sector of each partition is a boot sector and contains a BIOS parameter block.

The partitions are described by the 12 byte structure above.



Optional partitions

The bad sector list is usually held at the end of the device. If the parameter bsl_cnt is zero, there are no bad sectors.

Chapter 3
Atari ST traps and utilities

General

The operating system (TOS) is a mixture of GEM and an Atari OS, both can completely control the system but the programmer is advised to use the GEM host facilities, although calls to the various types of utilities can be mixed. There are many reasons for using a consistent set of calls, not the least being that the programmer can write programs which are portable to other computers that contain the same operating systems. Although the writers present intention may not be to provide the program on an alternative computer system, it is wise to adhere preferably to GEM calls if possible. Those who have programs generated on older 8-bit machines, and now find that they cannot be used, will understand the need for portability.

The BIOS and BDOS use and preserve registers in a rather haphazard manner, the following may be used as a guide:

$13 BIOS calls	preserve d3-d7/a3-a7
$14 Extended BIOS calls	preserve d3-d7/a3-a7
$ 1 BDOS calls	use d0-d4/a0-a4

The A-line routines provide access to the graphic primitives; they will not produce portable code but will give very rapid execution of graphic functions.

Lastly a word of warning. GEM was developed for use on the IBM PC, and as such ran on the Intel 8086 processor, which stores addresses in memory low byte first. Atari GEM uses the same convention in some of the tables and parameter blocks, it is a point programmers should be aware of, as a mixture of conventions of this kind is likely to cause problems.

Traps

GEM BIOS CALLS

To access the GEM BIOS functions, push the parameters in the order given onto the stack and then call Trap#13. Reply or status is returned in register D0 and the data placed on the stack trashed.

Typical use might be:

```
move.w driveA,-(sp)     * push device code
move.w record,-(sp)     * push record to start
move.w count, -(sp)     * push no. of sectors
move.l addrss,-(sp)     * push buffer address
move.w    #0, -(sp)     * push read data
move.w    #4, -(sp)     * push rwabs function call
trap   #13              * call the function
add.w     #14,sp        * tidy the stack
tst.w  D0               * test for error
rts                     *
```

It is the programmers responsibility to tidy the stack after the call. The BIOS, accessible from user mode, is re-entrant to three levels of calls, users are advised that this non-standard feature should be used wisely where program portability is required.

Gem BIOS calls

Function	*Parameters to push onto stack*	*Notes*
pmpb.L:	Pointer to empty memory parameter block to be filled — MPB structure: p20	Init vals:
	Memory_free_list →	MD in BSS
	Memory_alloc_list →	0
	Roving_pointer →	MD in BSS
see Appendix F	MD structure: →	
	Next_link_MD →	0
MD = memory descriptor	Start_addr_block →	mbottom
	No_bytes_block →	mtop-mbot
	Owner_description →	0
GETMPB 0: (#$00) Trap#13	Get/fill a memory parameter block (Tidy #6) (No return)	
dev.W:	dev code range 0 printer – parallel range 1 aux–RS232 2 con – screen 3 – midi 4 – keyboard	Operations 0 and 4 are illegal in this mode.
No range error checking		
BCONSTAT 1: (#$01) Trap#13	return char_device input status (Tidy #4) #$FFFF yes (D0.L)	Return D0.L #$0000 none
dev.W:	device code 0 to 3 (if dev 2, also ret IBM-PC compatible code in highword)	WAIT for a character
BCONIN 2: (#$02) Trap#13	input character from device (Tidy #4) (ASCII D0.W)	D0.L reply.
char.W:	character to be sent *(see Appendix C*	Wait until
dev.W:	device code 0 to 4 *for escape codes)*	character
BCONOUT 3: (#$03) Trap#13	output character to device (Tidy #6) (No return)	sent.

Gem BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
driv.W:	device code 0 = floppy A 1 = floppy drive B 2+ = disks, networks etc.	0 ret ok neg error
recn.W:	logical sector number to start at	rd/wr mode
secn.W:	number of sectors to transfer	2 & 3 allow
buf.L:	buffer address (very slow if odd)	formatter
rwfl.W:	rd/wr flag 0 read 1 write 2 read do not affect 3 write media-change	to read & write and allow BIOS to recognize
RWABS 4: (#$04) Trap#13	rd/wr logical sectors on a device (Tidy #14) (D0.L)	formatted disk media change
vec.L:	vector slot address (–1L no change)	0 to FF sys
vecn.W:	vector number to set/get	to $1FF GEM
SETEXC 5: (#$05) Trap#13	set exception vector *(see below)* (Tidy #8) (D0.L)	to $FFFF OEMs
TICKCAL 6: (#$06) Trap#13	return system elapsed time mS (Tidy #2) (D0.L)	
driv.W:	device code (0 to 2+, as rwabs)	Boot on $446
GETBPB 7: (#$07) Trap#13	Get BIOS parameter block pointer (Tidy #4) (D0.L)	D0.L = add. 0 = not found
dev.W:	device code – as per bconstat 0 to 4	0 = not ready
BCOSTAT 8: (#$08) Trap#13	Return device character output status (Tidy #4) (D0.L)	–1 = ready to send
driv.W:	device code (0 to 2+, as per rwabs)	
MEDIACH 9: (#$09) Trap#13	Get media status 0 = Media no change 1 = Media maybe changed 2 = Media has changed (Tidy #4) (D0.L)	GEMDOS will try to read media with a status value of 1
	must be updated by installable drives	Bits 0–31
DRVMAP 10: (#$0A) Trap#13	Get bitmap of drives (Tidy #2) (D0.L)	1 = drive in 0 = drive out

Gem BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
mode.W:	Mode bits	If mode –ve
	7 reserved (zero)	get IBM-PC
	6 left mouse button (insert)	state of
Note: Not all GEMs	5 right mouse button (clr/home)	SHIFT keys
read bits 5 & 6	4 CAPS LOCK as bit vector	
	3 ALT key	in D0.L low
	2 CONTROL key	byte.
	1 left SHIFT key	Critical
	0 right SHIFT key	code for
KBSHIFT 11:	Set keyboard shift bits	portability
(#$0B) Trap#13	(Tidy #4) (old shift bits D0.L)	

The extended GEMDOS vectors (Appendix A) may be employed by user programs but should take note of the following:

$100 etv_timer: Word value on stack is number of millisecs since last tick. Save all registers

$101 etv_critic: Stack word value is error number, save registers used.
To ignore an error set D0.L = 0
To retry an error set D0.L = $10000
To abort an error set D0.L = sign extend stack parameter.

$102 etv_term: Abort termination by a longword jump back to the top of the calling application or terminate via an RTS

EXTENDED BIOS CALLS

To access the extended BIOS functions, push the parameters in order given onto the stack and then call trap#14 from user or supervisor mode. Reply or status is returned in register D0.

Typical use might be:

```
move.l vector,-(sp)     * push vector address
move.l parblk,-(sp)     * push parameter block address
move.w type,  -(sp)     * push type of mouse action
move.w    #0, -(sp)     * push initmouse call
trap   #14              * call the function
add.w     #12,sp        * tidy the stack
tst.w  D0               * test for error
rts
```


Extended BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
vect.L:	vector address (mouse interrupt handler)	If mode = 2
para.L:	param 1_y = 0 top, 0_y = 0 bottom	then extra
(Block	block Mouse button command(#$07)	word sized
contains	address x param thresh/scale/delta	parameters
4 bytes)	y param thresh/scale/delta	required in
type.W:	mode 0 disable mouse	param block
	1 enable relative mouse	xmax
	2 enable absolute mouse	ymax
	3 unused	xinitial
	4 enable keycode mouse	yinitial
INITMOUS 0:	Initialize mouse packet handler	See call 34
(#$00) Trap#14	(Tidy #12) (No return)	re vect address
numb.W:	Bytes from memory top to be saved	MUST call
SSBRK 1:	Reserve block of memory at high RAM	before OS
(#$01) Trap#14	(Tidy #4) (Return D0.L)	initialized
_PHYSBASE 2:	Get screen base address (physical)	At next
(#$02) Trap#14	(Tidy #2) (Return D0.L)	vblank
_LOGBASE 3:	Get screen logical base address now	Used by GSX
(#$03) Trap#14	(Tidy #2) (Return D0.L)	on screen
_GETREZ 4:	Get screen resolution	Either 0, 1
(#$04) Trap#14	(Tidy #2) (Return D0.W)	or 2
rez.W:	Set screen resolution (0, 1 or 2)	Negative
	clr scrn, home cursor, reset VT52	parameters
ploc.L:	Set screen physical location (next vblank)	are ignored
lloc.L:	Set screen logical location (now)	so a single
_SETSCREEN 5:	Set screen parameters	parameter
(#$05) Trap#14	(Tidy #12) (No return)	can be set
palp.L:	Set pallette pointer (word boundary	At next
_SETPALLETE 6:	Set pallette hardware register contents	vblank
(#$06) Trap#14	(Tidy #6) (No return)	
colr.W:	Set colour (format-16 bit color word)	If colour
coln.W:	Set color number (0 to 15)	negative
_SETCOLOR 7:	Set a color in hardware pallette	ignore.
(#$07) Trap#14	(Tidy #6)	Return old
		color (D0.W)

Extended BIOS calls – continued

Function		*Parameters to push onto stack*	*Notes*
	secn.W:	number of sectors to be read	Ret D0.W = 0
	sidn.W:	side number selected	for ok
	trkn.W:	track number to seek to	else failed
	stsc.W:	sector to start reading from (1to9)	error number
	devn.W:	floppy device number (0 or 1)	
	scrt.L:	#0, not used at present.	
	buff.L:	word aligned sized buffer address →	must be big
_FLOPRD	8:	Read sectors from a floppy drive	enough
(#$08)	Trap#14	(Tidy #20)	
	secn.W:	number of sectors to write (< = sectors/track)	Ret D0.W = 0
	sidn.W:	side number selected	for ok
	trk.nW:	track number to seek to	else failed
	stsc.W:	sector to start writing to (1–9)	error number
	devn.W:	floppy device number (0 or 1)	
	scrt.L:	#0, not used at present.	Writing to
	buff.L:	word aligned buffer address	boot 1,0,0
_FLOPWR	9:	Write sectors to a floppy drive	sets 'maybe'
(#$09)	Trap#14	(Tidy #20)	media changed
	fcod.W:	$E5E5 format code (not 0 or FxFx)	Ret D0.W = 0
	magc.L:	$87654321	for ok
	intl.W:	Sector interleave factor (say 1)	else failed
	sidn.W:	side number to format (0 or 1)	error number
	trkn.W:	track number to format (0 to 79)	Buffer holds
	sptk.W:	Number of sectors/track to format (say 9)	0 terminatd
	devn.W:	floppy device number (0 or 1)	list of bad
	scrt.L:	#0, not used at present.	sectors.
	buff.L:	word aligned buffer addr (8K, 9track)	Formatting
_FLOPFMT	10:	Format a floppy disk	sets
(#$0A)	Trap#14	(Tidy #26)	media changed
GETDSB	11:	Get device status block pointer	Obsolete
(#$0B)	Trap#14	(Tidy #2) (Return D0.L)	function.
	ptr.L:	Pointer to character vector	
	cnt.W:	Number of characters to write less one.	
MIDIWS	12:	Write a string to midi port	
(#$0C)	Trap#14	(Tidy #8) (No return)	

Extended BIOS calls – continued

Function		*Parameters to push onto stack*		*Notes*
	vect.L:	Address of interrupt routine		Old vector
	intn.W:	Interrupt number (0 to 15)		is lost.
_MFPINT	13:	Set MFP interrupt		
(#$0D)	Trap#14	(Tidy #8)	(No return)	
	devn.W:	Serial dev	0: RS232	For RS232
		1: Keyboard		identical
		2: Midi		o/p buffer
Return a pointer .L to a serial devices input buffer record param block (brpb)		L. pointer to dev buffer		follows i/p
		W. size of buffer		
		W. head index		High & low
		W. tail index		w'mark start
		(W. low-water mark		RS232 xon
		(W. high-watermark		xoff if
IOREC	14:	Get pointer to serial device i/p brpb		flow control
(#$0E)	Trap#14	(Tidy #4)	(Return D0.L)	enabled.
	scr.W:	Sync char reg	68901	–1 params
	tsr.W:	Tx status reg	MFP register	do not
	rsr.W:	Rx status reg	settings	change
	usr.W:	Usart cntrl reg	*(Chapter 1)*	registers
	flow.W:	0 No flow control (default)		
		1 xon/xoff (∧S/∧Q)		
		2 RTS/CTS		
		3 xon/xoff & RTS/CTS		
	baud.W:	0 = 19200	6 = 1800	12 = 134
		1 = 9600	7 = 1200	13 = 110
		2 = 4800	8 = 600	14 = 75
		3 = 3600	9 = 300	15 = 50
		4 = 2400	10 = 200	
		5 = 2000	11 = 150	
_RSCONF	15:	Configure RS232 port		
(#$0F)	Trap#14	(Tidy #14)	(No return)	
	capl.L:	Caps lock	Set pointers to 128 byte	Ret pntr to
	shft.L:	Shift	Keyboard translation	structure:
	unsh.L:	Unshifted	table.	Unshift_table
_KEYTBL	16:	Set/get keyboard translation table pointer		Shift_table
(#$10)	Trap#14	(Tidy #14)	(D0.L)	Capslk_table

Extended BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
	Bit zero poor distribution	
_RANDOM 17:	Get 24-bit pseudo random number	Bits 24–31
(#$11) Trap#14	(Tidy #2) (D0.L)	are zero
	exfl.W: 1 = boot sector executable	–1 retains
	0 = non-executable boot sector	old values.
	dskt.W: 0 = 40 track SS	
	1 = 40 track DS	
	2 = 80 track SS	
	3 = 80 track DS	Image is
	sern.L: random boot serial no. if> = #$01000000	written
	buf.L: pointer to any 512-byte buffer	to volumes
_PROTOBT 18:	Prototype a boot sector image	boot sector
(#$12) Trap#14	(Tidy #14) (No return)	
	secn.W: number of sectors to verify (< = sectors/track)	Ret D0.W = 0
	sidn.W: side number selected	for ok
	trkn.W: track number to seek to	else failed
	stsc.W: sector to start reading from (1to9)	error number
	devn.W: floppy device number (0 or 1)	
	scrt.L: #0, not used at present.	Buffer holds
	buff.L: word aligned 1024 byte buffer address	0 terminatd
_FLOPVER 19:	Verify sectors from a floppy drive	list of bad
(#$13) Trap#14	(Tidy #20)	sectors.W
SCRDMP 20:	Dump screen to printer	At present
(#$14) Trap#14	(Tidy #2) (No return)	mono only.
	rate.W: Rate = 1/2 cycle time–1 retains	
	60/50 Hz color, old values 70 Hz monochrome	
	attr.W: 0_Hide cursor 4_Set rate	
	1_Show cursor 5_Get rate	Returns
	2_Blink cursor 6_unused	old rate high
	3_Noblink cursor 7_unused	old attrib low
CURSCON 21:	Set/get cursor blink	word byte.
	rate & attributes	
(#$15) Trap#14	(Tidy #6) (Return D0.W)	

Extended BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
	date.L: 32-bit DOS format date and time	Date Hiword
SETTIME 22:	Set ikbd time and date	Time Loword
(#$16) Trap#14	(Tidy #6) (No return)	
GETTIME 23:	Get ikbd 32-bit format date & time	
(#$17) Trap#14	(Tidy #2) (D0.L)	
BIOSKEY 24:	Restore power up keyboard setting	Reset translation
(#$18) Trap#14	(Tidy #2) (No return)	tables
	pntr.L: Pointer to character string vector	Send cmd
	nch.W: Count of characters to send –1	to ikbd
IKBDWS 25:	Write a string to intelligent keyboard	
(#$19) Trap#14	(Tidy #8) (No return)	
	intn.W: MK68901 interrupt number	
JDISINT 26:	Disable a MK68901 interrupt	
(#$1A) Trap#14	(Tidy #4) (No return)	
	intn.W: MK68901 interrupt number	
JENABIN 27:	Enable a MK68901 interrupt	
(#$1B) Trap#14	(Tidy #4) (No return)	
	regn.W: PSG register number (00 to 0FH)	regn ORed
	data.B: Byte to write to register	#$00 read
GIACCES 28:	Read/write a sound chip register	#$80 write
(#$1C) Trap#14	*Atomic access only* (Return D0.B)	
	bitn.W: Bit number to be set	
OFFGIBT 29:	Atomically set Port A bit to zero	
(#$1D) Trap#14	(Tidy #4) (No return)	
	bitn.W: Bit number to be set	
ONGIBIT 30:	Atomically set PORT A bit to one	
(#$1E) Trap#14	(Tidy #4) (No return)	

Extended BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
	vec.L: Pointer to an interrupt handler data.W: Byte placed in timer's data register cntl.W: Timers control register setting timr.W: Timer number allocations are: 0_A Res'd for end-users & applications 1_B Reserved for graphics primarily 2_C System timer (GEM etc) 3_D RS232 baud rate and mere users	
XBTIMER (#$1F) Trap#14	31: Provide control timing facility (Tidy #12) (No return)	
	ptr.L: Pointer to table of bytes (command data) cmd 0 to 15 load reg 0–15 with data cmd 128 load tempreg with databyte cmd 129 reg # to load using tempreg twos c value to add to tempreg terminate on tempreg value cmd 130–255 set delay data (ticks)	Usually in twos except 129 which is in sets of 4 bytes. → 0 = stop
DOSOUND (#$20) Trap#14	32: Produce a sound (Appendix L) (Tidy #6)	
	conf.W: Bit 0 0 = dot matrix, 1 = daisy wheel 1 0 = colour dev, 1 = monochrome 2 0 = Atari prnt, 1 = Epson prntr 3 0 = draft, 1 = final 4 0 = parallel, 1 = RS232 port 5 0 = formfeed, 1 = single sht 6–14 reserved 15 must be zero	–1 returns configuration byte else change and return the old value.
SETPRT (#$21) Trap#14	33: Get/set printer configuration byte (Tidy #4) (D0.W)	

Extended BIOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
Structure longword format	MIDI_input (BIOS buffer routine) keybrd_err / MIDI_err: Called when over-run detected ikbd_stat, mouse_pack, clock_pack, joyst_pack, MIDI_vec, ikbd_vec: Pointer to packet handlers (pointer to packet received in A0 & on stack.L) Call when character available on 6850	→ D0.B char 68901 or 6850s (mouse vect used by GEM & GSX.)
KBDVBAS (#$22) Trap#14	34: Return pointer to structure base (Tidy #2) (D0.L)	Return by RTS and within 1ms
	rept.W: Rate of key-repeats (System ticks) init.W: Delay before key-repeat starts	–1 params no change.
KBRATE (#$23) Trap#14	35: Get/set keyboard repeat rate (Tidy #6) (D0.W)	Init hibyte rept lobyte
_PRTBLK (#$24) Trap#14	36: (Tidy #2)	
VSYNC (#$25) Trap#14	37: Wait till next vblank and return (Tidy #2) (No return)	Graphics synchronize
	Code.L: Pointer to code that ends with RTS (Hackers' access to hardware & protected locations	Must not call BIOS or GEMDOS functions
SUPERX (#$26) Trap#14	38: Exec code in supervisor mode (Tidy #2)	
PNTAES (#$27) Trap#14	39: If AES not present then return, else reboot (Tidy #2)	

GEM BDOS FUNCTION CALLS

To access GEM BDOS functions, push the parameters in the order given onto the current stack and then call trap#1. Any byte, word or longword reply or the address of a parameter block will be returned in register D0.

```
move.W driveB,-(SP)   * push drive number (2)
move.W  #13,-(SP)     * push setdrv function call
trap    #1            * call the function
add.W   #4, SP        * tidy stack
rts                   * return with bitmap in D0
```

It is the programmer's responsibility to maintain the stack integrity after the call.

Function	*Parameters to push onto stack*	*Notes*
P_TERM_OLD (#$00) Trap#1	0: End process and return to parent. (Tidy #2)	Return code zero
C_CONIN (#$01) Trap#1	1: Read character from standard i/p & echo (Tidy #2) (Return D0.L)	
C_CONOUT (#$02) Trap#1	char.W: Character to be printed 2: Write character to standard output (Tidy #4) (No return)	← The console scan code is returned in the low byte of the high word.
C_AUXIN (#$03) Trap#1	3: Read character from auxiliary port (Tidy #2) (Return D0.L)	The upper byte of the word sent must be 0 for future compatibility.
C_AUXOUT 4: (#$04) Trap#1	char.W: Character to be printed Write character to standard aux device (Tidy #4) (No return)	
C_PRNOUT 5: (#$05) Trap#1	char.W: Character to be printed Write character to standard print device (Tidy #4) (No return)	
C_RAWIO 6: (#$06) Trap#1	parm.W: If parm = 255 (00FF) then read else parm is character to be written Raw I/O to standard input/output (Tidy #4) (Return D0.L)	If no char then D0.L = 0
C_RAWCIN 7: (#$07) Trap#1	Raw input from standard input (Tidy #2) (Return D0.L)	No echo. Pass controls

Gem BDOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
C_NECIN 8: (#$08) Trap#1	Read a character from standard input (Tidy #2) (No return)	No echo. ∧C, ∧Q & ∧S act
C_CONWS 9: (#$09) Trap#1	addr.L: Address of null terminated string Write string to standard output (Tidy #6) (No return)	Char bytes terminated by a zero.
C_CONRS (#$0A) Trap#1	addr.L: Address of input buffer (First byte data portion length) 10: Read edited string from standard input (Tidy#6) (Buffer returns)	On return, 2nd len read 3–n chars n + 1 zero
C_CONIS (#$0B) Trap#1	11: Check status of standard input (Tidy #2) (D0.L)	character ready –1_Yes, 0_No
D_SETDRV (#$0E) Trap#1	driv.W: Drive number: 0 = A, 1 = B . . 15 = P 14: Set default drive (Tidy #4) (D0.L)	Return bitmap of drives present
C_CONOS (#$10) Trap#1	16: Check status of standard output (Tidy #2) (D0.L)	–1 ready to Rx. 0 = not
C_PRNOS (#$11) Trap#1	17: Check status of standard print device (Tidy #2) (D0.L)	–1 ready to print, 0 = not
C_AUXIS (#$12) Trap#1	18: Check status of standard aux device i/p (Tidy #2) (D0.L)	–1 char rx 0 = no chars
C_AUXOS (#$13) Trap#1	19: Check status of standard aux device o/p (Tidy #2) (D0.L)	–1 ready to Rx. 0 = not
C_GETDRV (#$19) Trap#1	25: Get current drive (Tidy #2) (D0.L)	drive A = 0 B = 1 etc.
F_SETDTA (#$1A) Trap#1	addr.L: Disk transfer address 26: Set disk transfer address (Tidy #6) (No return)	Address used by f_sfirst (78)
T_GETDATE (#$2A) Trap#1	42: Get date (as *set date* format) (Tidy #2) (D0.L)	Date return in low word

Gem BDOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
	date.W: Date format date: bits 0–4, 1 to 31	Error ret'd
T_SETDATE	43: Set date: month: bits 5–8, 1 to 12	if date not
(#$2B) Trap#1	(Tidy #4) year: bits 9–15, 1980–2100	valid
T_GETTIME	44: Get time (as *set time* format)	Time return
(#$2C) Trap#1	(Tidy #2) (D0.L)	in low word
	time.W: Time format secs: bits 0–4, step 2s	Error ret
T_SETTIME	45: Set date: mins: bits 5–10	if date not
(#$2D) Trap#1	(Tidy #4) hour: bits 11–15 (D0.L)	valid
F_GETDTA	47: Get disk transfer address	
(#$2F) Trap#1	(Tidy #2) (D0.L)	
S_VERSION	48: Get version no. (1.00 lo-hi byte)	0001_H for
(#$30) Trap#1	(Tidy #2) (D0.W)	1st release
	exit.W: Exit code (process return code) *total size of program, base page, text, bss etc*	May cause
	keep.L: # bytes to keep in process description	problems
P_TERMRES	49: Terminate and stay resident	for future
(#$31) Trap#1	(Tidy #8) (No return)	conversions
	driv.W: Drive number: 0 = current, 1 = A, 2 = B	Buffer pb.L
	info.L: Address of drive info buffer	# free clust
D_FREE	54: Get drive free space (data in buffer	#clust–total
	4 × longwords)	#bytes/sect
(#$36) Trap#1	(Tidy #8) (No return)	#sect/clust
	path.L: Address of string containing pathname	
D_CREATE	57: Create a subdirectory	
(#$39) Trap#1	(Tidy #6) (D0.L)	Pathname is
	path.L: Address of string containing pathname	terminated
D_DELETE	58: Delete a subdirectory	in a null.
(#$3A) Trap#1	(Tidy #6) (D0.L)	0 ret ok
	path.L: Address of string containing pathname	neg error
D_SETPAT	59: Set current directory	
(#$3B) Trap#1	(Tidy #6) (D0.L)	

Gem BDOS calls – continued

Function	*Parameters to push onto stack*	*Notes*
	attr.W: File attributes: #$01 read only	Ret file
	#$02 hidden file, #$04 hidden system file	handle if
	#$08 File, vol label in 1st 11 bytes	ok. Neg if
	path.L: Address of string containing pathname	error.
F_CREATE	60: Create a file	Pathname
(#$3C) Trap#1	(Tidy #8) (D0.L)	ends in 0
	attr.W: File read-write mode	Ret file
	0 = file open for read only	handle if
	1 = file open for write only	ok. Neg if
	2 = file open read and write	error.
	path.L: Address of string containing pathname	
F_OPEN	61: Open file	Pathname
(#$3D) Trap#1	(Tidy #8) (D0.L)	ends in 0
	hndl.W: File handle (errors may crash system)	0 ret ok
F_CLOSE	62: Close file	neg error.
(#$3E) Trap#1	(Tidy #4) (D0.L)	
	buff.L: Address of buffer to store bytes	D0 contains
	byts.L: Number of bytes to read	no. bytes
	hndl.W: File handle (errors may crash system)	read.
F_READ	63: Read file	Negative on
(#$3F) Trap#1	(Tidy #12) (D0.L)	error.
	buff.L: Address of buffer storing bytes	D0 contains
	byts.L: Number of bytes to write	no. bytes
	hndl.W: File handle (errors may crash system)	written.
F_WRITE	64: Write file	Negative on
(#$40) Trap#1	(Tidy #12) (D0.L)	error.
	path.L: Address of string containing pathname	0 ret ok
F_DELETE	65: Delete file	neg error.
(#$41) Trap#1	(Tidy #6) (D0.L)	
	fmod.W: 0: move n bytes from beginning	Pos moves
	1: move n bytes from current posn	to end of
	2: move n bytes from end of file	file, neg
	hndl.W: File handle	beginning
	nbyt.L: Signed number of bytes argument	
F_SEEK	66: Seek file pointer	D0 = Abs. file
(#$42) Trap#1	(Tidy #10) (D0.L)	pointer loc.

Gem BDOS calls – continued

Function		*Parameters to push onto stack*	*Notes*
	attr.W:	File attributes: #$01 read only $02 hidden file, #$04 hidden system file, $08 File, volume label in 1st 11 bytes, $10 File is a subdirectory, $20 File has been written & closed,	Ret file handle if ok. Neg if error
	wrt.W:	0_get/1_set file attributes	Pathname is terminated in a null.
	path.L:	Address of string containing pathname	
F_ATTRIB	67:	Get/set file attributes	
(#$43) Trap#1	(Tidy #10)	(D0.L)	Get in D0.L
	shnd.W:	Standard file handle to duplicate	
F_DUP	69:	Duplicate file handle	Error ret
(#$45) Trap#1	(Tidy #4)	(D0.L)	
	shnd.W:	Standard file handle to force/0 con i/p	–1 con o/p
	nhnd.w:	Non-standard file handle	–2 serial
F_Force	70:	Force point file handle to non-standard handle file or device	–3 parallel
(#$46) Trap#1	(Tidy #6)	(D0.L)	
	driv.W:	Drive number: 0 = default, 1 = A... etc.	Buffer min 64 bytes.
	path.L:	Address of 64 byte buffer for pathname	
D_GETPATH	71:	Get current directory	
(#$47) Trap#1	(Tidy #8)	(D0.L)	
		allocated block may not be on a word boundary	
	nbyt.L:	Bytes to allocate or –1 ret max available	D0.L = 0 if alloc fails or pointer to block.
M_MALLOC	72:	Allocate memory (D0.L start pointer)	
	or	Read free memory (D0.L bytes available)	
(#$48) Trap#1	(Tidy #6)	(D0.L)	
	frad.L:	Address of memory to free	0 ret ok
M_FREE	73:	Free allocated memory	neg error
(#$49) Trap#1	(Tidy #6)	(D0.L)	
	rmem.L:	Length of retained memory	Reallocates unused mem. for GEMDOS.
	mmem.L:	Start of memory space to modify	
	zero.W:	zero	
M_SHRINK	74:	Shrink size of allocated memory	0 ret ok
(#$4A) Trap#1	(Tidy #12)	(No return)	neg error

Gem BDOS calls – continued

Function		*Parameters to push onto stack*	*Notes*
	penv.L:	Pointer to environ string, 0 for parent	Mode 3 is used for overlays,
	pcmd.L:	Pointer to command tail incl redirection	
	path.L:	Address of string containing pathname	
	mode.W:	0 = load & exececute ret term child code 3 = load only. ret. D0.L base page add. 4 = create basepage, 5 = execute only	Ret D0.L error if load fails.
P_EXEC	75:	Load or execute a process	
(#$4B) Trap#1	(Tidy #16)	(D0.L)	
		set return code positive to avoid confusion with negative system error codes	
	stat.W:	Interrogation code for parent	0 ret ok non-0 error
P_TERM	76:	Terminate process, control to parent	
(#$4C) Trap#1	(Tidy #4)	(D0.L)	
	satt.W:	Search attributes $00 normal files, #$01 read only $02 hidden files, #$04 hidden system file $08 volume label file, #$10 subdir files $20 File has been written & closed	Filename may include '*' or '?' wildcards.
	path.L:	Address of string containing pathname	If file not found ret EFILNF code in D0.L
F_SFIRST	78:	Search for 1st occurence filespec 44-byte DTA buffer created if found 0–20 o/s reserved 21 file attributes 22–23 Time stamp 24–25 Date stamp 26–29 Filesize.L 30–43 Name.ext	
(#$4E)			
(#$4E) Trap#1	(Tidy #8)	(D0.L)	
	79:	Search for next occurence filespec (Uses 1st 20 bytes of DTA buffer, name.ext updated on success)	1st 20 bytes DTA buffer must not be altered.
F_SNEXT			
(#$4F) Trap#1	(Tidy #2)	(D0.L)	
	pth2.L:	Pointer to 'new' file string	Rename a file
	pth1.L:	Pointer to 'old' file string	
	zero.W:	zero	
F_RENAME	86:	Rename a file	
(#$56) Trap#1	(Tidy #12)	(D0.L)	

Gem BDOS calls – continued

Function		*Parameters to push onto stack*	*Notes*
	info.W:	0_set/1_get date and time	
	hndl.W:	File handle	
	buff.L:	Time and date buffer pointer	
F_DATIME	87:	Get/set file date and time stamp	
		Buffer first word	
		Bit format:	
		days 0–4, 1 to 31	
		month 5–8, 1 to 12	
		year 9–15, 1980 to 2100	
		Buffer second word	
		Bit format:	
		secs 0–4 in 2 second steps	
		mins 5–10	
		hour 11–15	
(#$57) Trap#1	(Tidy #10)		(No return)

DTA buffer

Offset			
$00	0		OS reserved
$15	21	Byte	file attributes
$16	22	Word	file time stamp
$18	24	Word	file date stamp
$1A	26	Longword	longword file size
$1E	30	7 words	name and ext of file found

Use function #$1A (dec 26) to set DTA buffer address and function #$2F (dec 47) to get DTA address.

Supervisor/User toggle

This special function allows users to get in and out of supervisor mode from GEM DOS.

Function		*Parameters to push onto stack*		*Notes*
	stck.L:	–1_get mode:	Return 0_user (D0.L) 1_supervisor	
		<>–1 switch mode		Return
		a) User to supervisor mode		value of
		0_set supervisor stack equal		old super
		to user stack before call		stack in
		<>0_set supervisor stack equal		D0.L
		to stck.L		
		b) Supervisor to user mode		The old
		set supervisor stack from		value of
		stck.L which must be first		super stack
		SMODE function call or the		MUST be
		system will crash.		restored on
SMODE	32:	Set/get supervisor/user mode		process
(#$20) Trap#1	(Tidy #6)		(D0.L)	termination

Test for mode

```
move.L  #$1,-(sp)        *   Returns D0.L
move.W  #32,-(sp)        *     $0 = user mode
trap    #1               *     $FF = supervisor mode
addq    #6,sp            *
```

User to supervisor mode

```
clr.L   -(sp)            *   Set supervisor stack equal to
move.W  #32,-(sp)        *   user stack before this call,
trap    #1               *
addq    #6,sp            *
move.L  D0,save_stk      *   Save old supervisor stack value
```

Supervisor to user mode

```
move.L save_stk,-(sp)    *   Recover old supervisor stack
move.w  #32,-(sp)        *
trap    #1               *   and back into user mode.
addq    #6,sp            *
```

EXTENDED BDOS FUNCITON CALLS

To access the extended BDOS functions, the D0.W register is loaded with the function code, an address pointer is placed in D1 (.L) and trap #2 called. A return, if any, is placed in D0.W.

GEM VDI and AES may be accessed by loading the relevant parameter block address into d1, the function number into d0 and making an extended BDOS call:

GEM VDI

```
move.l  #contrl, pblock
move.l  #pblock,d1        * address of VDI param block
move.w  #$73,d0           * set d0 equal to 115 and
trap    #2                * execute an extended BDOS call
```

and GEM AES

```
move.l  #control,_c
move.l  #_c,d1            * address of AES param block
move.w  #$c8,d0           * set d0 equal to 200 and
trap    #2                * execute an extended BDOS call
```

Code#	*Hex*	*Dec*	*Function*	*Notes*
D0.W :	#$00	0		The function
RESET :			Terminate current program and	does not return
Trap #2:			return to CCP level	to calling program
D1.L :	#pblock		VDI param block pointer	
D0.W :	#$73	115	VDI function number	
Trap #2:			GEM VDI access	
D1.L :	#control		AES param block pointer	
D0.W :	#$C8	200	AES function number	
Trap #2:			GEM AES access	
D0.W :	#$C9	201		

The trap #2 reset call simply calls the GEMDOS trap#1 process terminate function #$4C.

GEM VDI function calls

The VDI functions are accessed through an extended BDOS call and the VDI parameter block (five longword pointers to the word tables; cntrl, input attribute and points, output attribute and points). The parameter and array blocks, which are usually initialized by an AES call to APPL_INIT, have the following formats:

VDI parameter block

Offset		D0
$ 0	Control table pointer	
$ 4	I/P attribute table pointer	intin
$ 8	I/P points table pointer	ptsin
$ C	O/P attribute table pointer	intout
$10	O/P points table pointer	ptsout
$14		

Control table

Offset		
$0	Opcode	
$2	Length of input coordinate table	Length in word pairs
$4	Length of output coordinate table	
$6	Length of input attribute table	Length in words
$8	Length of output attribute table	
$A	Subfunction ident number	
$C	Device handle	zero if can't be opened
($E→)	Opcode dependant information	

Attribute table

intin / intout Offset		Typical usage
$0		device id
$2		line type
$4		line colour
$6		mark type
$8		mark colour
$A		font

Points table

ptsin / ptsout Offset		Typical usage	
$0		x coordinate	word pair
$2		y coordinate	
$4			
$6			
$8		width	word pair
$A		height	

Not all the GEM VDI function calls have been implemented on the Atari ST, but they are listed in this section. It might help a little in translating programs to the ST that use these function calls. A minimum application stack space of 128 bytes is required, plus space for the GEM arrays. The VDI function calls have been detailed in groups as follows:

Workstation control functions:
Define the workstation parameters and defaults; these govern the font and the window size to be used and the generation of virtual screens.

Output functions:
These functions draw the graphic primitive on the specified output device.

General drawing primitive functions:
Define the basic graphic primitives of line, arc, filled and unfilled ellipse and rectangle, and of justified text.

Attribute functions:
Define the output style of the graphic primitives; the line, marker, text cell and polygon for colour, size and fill.

Raster operations:
Provide the ability to transpose a source block of pixels to a destination location on the basis of a logical operation between the bits comprising the source and destination.

Input functions:
Enable the programmer to provide the user with both a 'request and wait on event' and a 'request, sample and return' mode of inquiry.

Inquire functions:
Return the status or attributes of a specific device

Escape functions:
Enable the application to access special features applicable to certain graphic devices.

VDI Parameter block sizes

The numbers of parameters required by the various functions are detailed in the tabular format:

Control table

Function	*Op*	*Pointpair in*	*Pointpair out*	*Integers in*	*Integers out*	*GDP*	*Dev. name*	*Comments*
	$0	*$2*	*$4*	*$6*	*$8*	*$A*	*$C*	

The table contains details of the parameter input and output word sizes; note that the points value is half the table size (a point is defined by a pair of x and y word-sized coordinates – a longword).

OPEN WORKSTATION FUNCTION v_opnwk
The major VDI function in terms of size is the 'open workstation function', which sets up a named screen (device handle); the desktop window is identified as device name zero. The new screen is initialized to graphics mode, cleared and the parameter table outputs initialized. The v_opnwk (op_1) function **is not available** on the Atari ST, programmers should use the virtual workstation function v_opnvwk (op_100).

The control table

Control offset		*Array Data size B*		*Function*	
$0	0	1		Opcode for 'open workstation	
$2	2	0	0	# of input point pairs	ptsin
$4	4	6	24	# of output point pairs	ptsout
$6	6	11	22	Len of input attribute table	intin
$8	8	45	90	Len of output attribute table	intout
$A	10	—		Not used	
$C	12	x		Handle for this device	(out)

Attribute input table (intin)

Intin Offset		*Initial defaults (style, colour etc.) code*		*VDI Op no*
$ 0	0	Device driver	(screen = 1)	
$ 2	2	Linetype	(solid = 1)	15
$ 4	4	Polyline colour index		—
$ 6	6	Marker type	(dot = 1)	18
$ 8	8	Polymarker colour index		18
$ A	10	Text face		21
$ C	12	Text colour index		—
$ E	14	Fill interior style		23
$10	16	Fill style index		24
$12	18	Fill colour index		—
$14	20	NDC to RDC transform flag (2 only) 0 map full NDC to full RC 1 reserved 2 Use RC coords		

The input ranges required to open a workstation with a specific attribute can be found, in the table box for that attribute, later in this chapter.

The procedure names are limited to the maximum of eight unique characters supported by the Atari ST 'C' compiler. Note that 'C' external names are prefixed by an '_' (underscore) which reduces the uniqueness to seven characters.

Attribute output table (intout)

Intout Offset		*Default output parameters*		*Typical b&w values*	
$ 0	0	Max pixel width 0 to 639	mono	$27f	639
$ 2	2	Max pixel height 0 to 399	values	$18f	399
$ 4	4	Dev coord flag (0 = fine, 1 = coarse)			0
$ 6	6	Pixel height, microns			372
$ 8	8	Pixel width, microns			372
$ A	10	No. char heights (0 = continuous)			3
$ C	12	No. linetypes			7
$ E	14	No. line widths (0 = continuous)			0
$10	16	No. marker types			6
$12	18	No. marker sizes (0 = continuous)			8
$14	20	No. faces supported			1
$16	22	No. patterns		$18	24
$18	24	No. hatch styles		$c	12
$1A	26	No. simultaneous colours (2 = mono)			2
$1C	28	No. generalized drawing primitives		$a	10
		List of 1st 10 GDP's (–1 ends list)			
$1E–$30		1 = Bar	6 = Elliptical arc		
		2 = Arc	7 = Elliptical pie	1 to 10	
30–48		3 = Pie slice	8 = Rounded rectangle		
		4 = Circle	9 = Filled 8 above	3 0 3	
		5 = Ellipse	10 = Justified graphic text	3 3 0	
		Attrib list for GDP's		3 0 3	
$32–$44		0 = Polyline			2
50–68		1 = Polymarker	3 = Fill area	respectively	
		2 = Text	4 = None		
$46	70	Colour	0 = no, 1 = yes		0
$48	72	Text rotation	Capability		1
$4A	74	Fill area	flags		1
$4C	76	Cell array operation			0
$4E	78	No. colours (2 = mono, >2 = no. colours)			2
$50	80	No. locator devices	1 = keyboard only 2 = keyboard + i/p		2
$52	82	No. valuator devices	1 = keyboard		1
$54	84	No. choice devices	1 = function keys 2 = button device		1
$56	86	No. string devices	1 = keyboard		1
$58	88	Workstation type 0 = o/p only 1 = i/p only 2 = input/output	3 = reserved 4 = metafile output		2

Output points table (ptsout)

Ptsout Offset		*Output points table*	*Typical b & w values*	
$ 0	0	Min character width	5	
$ 2	2	Min character height	4	
$ 4	4	Max character width	7	
$ 6	6	Max character height	$d	13
$ 8	8	Min line width	1	
$ A	10	Zero	0	
$ C	12	Max line width	$28	40
$ E	14	Zero	0	
$10	16	Min marker width	$f	15
$12	18	Min marker height	$b	11
$14	20	Max marker width	$78	120
$16	22	Max marker height	$58	88

WORKSTATION CONTROL FUNCTIONS

The following functions set the workstation parameters and defaults for use by the application:

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Close workstation v_clswk *	2	0	0	0	0		—	Ret to alpha mode. Close device and flush buffers.
Open virtual screen v_opnvwk	100	0	6	11 parameters as opcode 1	45		i/p scrn o/p new window o.error	Permits multiple windows based on one screen with different attributes.
Close virtual screen v_clsvwk	101	0	0	0	0		—	Close virtual screens first. Stop further output to screen.
Clear workstation v_clrwk	3	0	0	0	0		—	Clear the screen. New page if poss. Del buffer data.
Update workstation v_updwk	4	0	0	0	0		—	Execute graphic commands waiting. No screen effect. Use to print data.
Load font vst_load_fonts	119	0	0	1 intin(0) = 0, reserved	1 intout(0) = # fonts loaded may not be implemented		—	Load additional fonts. intin(0) reserved for future use.
Unload font vst_unload_fonts	120	0	0	1 intin(0) = 0, reserved may not be implemented	0		—	Unload font from mem if no other live users. intin(0) reserved.
Set clipping rectangle vs_clip	129 a,b	2 c,d	0 intin(0) = 0_off <>0_on	1	0	(default)	—	Disable/enable clipping of output primitive. ptsin a,b,c,d

* Not available on the Atari ST, programmers should use v_clsvwk (op_101)

OUTPUT FUNCTIONS

The following functions draw the graphic primitives (lines, arc etc.) on the current device using the current attributes.

Output functions

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Polyline v_pline	6 minimum 2 coordinate pairs	n	0	0	0		—	Line join n pairs of points.
Poly-marker v_pmarker	7	n	0	0	0		—	Draw marker at each of n pairs of points.
Text v_gtext	8 intin(0) = text string ptsin(0) = x coor (2) = y coor	1	0	n	0 (Strg len) Text start position		—	Write char string to device. 0–255 Intin word LSB contains char.
Filled area v_fillarea	9	n n × x,y points	0	0	0		—	Outline if device can't fill. Close area if open.
Fill rectangle vr_recfl	114 a,b c,d	2	0 ptsin(0) = a (2) = b	0	0 ptsin(4) = c (6) = d		—	Rectangular area fill. ptsin a, b, c, d
Cell array v_cellarray	10 rect. array c,d a,b Row length Xc $ Words/row $ Rows Xc Writing mode	2 Cntrl $E Cntrl $10 Cntrl $12 Cntrl $14	0	n index	0 length of colour Colour index array Xc Yc		—	Draw rectangular cell array. ptsin a,b,c,d based on color cells Xc × Yc
Contour fill v_contour	103 intin(0) = color index ptsin(0) = x coor ptsin(2) = y coor	1	0	1	0 Starting point		—	Flood fill area bound by edge or colour.

GENERAL DRAWING PRIMITIVE FUNCTIONS (GDP's)

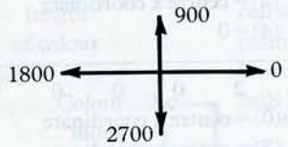
The GDP's provide the basic graphic primitives of line, arc, ellipse etc.

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
GDP (General format)	11	n	—	—	—	x	—	
Bar v_bar	11 ptsin(0) = corner x coordinate (2) = corner y coordinate (4) = diag opp x coor (6) = diag opp y coor	2	0	0	0	1	—	Area attributes
Arc v_arc	11 ptsin(0) = centre x coordinate (2) = centre y coordinate (4) = 0 (6) = 0 (8) = 0	4	0 ptsin(A) = 0 (C) = radius	2 intin(0) = start angle (2) = end angle	0 (E) = 0	2	—	Line attributes 0 to 3600
Pie v_pieslice	11 Parameters as for arc above	4	0	2	0	3	—	Area attributes
Circle v_circle	11 ptsin(0) = centre x coordinate (2) = centre x coordinate (4) = 0	3	0	0 ptsin(6) = 0 (8) = radius (A) = 0	0	4	—	Area attributes
Ellipse v_ellipse	11 ptsin(0) = centre x coordinate (2) = centre y coordinate (4) = radius x axis (6) = radius y axis	2	0	0	0	5	—	Area attributes
Elliptic arc v_ellarc	11 ptsin(0) = centre x coordinate (2) = centre y coordinate (4) = radius x axis (6) = radius y axis	2	0	2 intin(0) = start angle (2) = end angle	0	6	—	Line attributes 0 to 3600

General drawing primitive functions – continued

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
GDP (General format)	11	n	—	—	—	x	—	
Elliptic pie `v_ellpie`	11	2	0	2	0	7	—	Area attributes
	Parameters as for elliptic arc above							
Rounded rectangle `v_rbox`	11	2	0	0	0	8	—	Line attributes
	ptsin(0) = corner x coordinate (2) = corner y coordinate (4) = diagonally opposite x coordinate (6) = diagonally opposite y coordinate							
Filled rounded rectangle `v_rfbox`	11	2	0	0	0	9	—	Area attributes
	Parameters as for rounded rectangle							
Justified graphics text `v_justified`	11	2	0	2 + n	0	10	—	Text attributes
	ptsin(0) = x alignment (2) = y alignment (4) = string length (6) = zero			intin(0) = interword space flag (2) = interchar space flag 0 = don't mod space (4) = 1st char } Null terminated (4 + 2n) = Last char } string				
	Intin uses least significant byte for character							

Notation used for angular specifications:



ATTRIBUTE FUNCTIONS

The attribute functions determine the output style of all the graphic primitives; that is colour, line style, character size etc.

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Set writing mode `vswr_mode`	32	0	0	1	1 *		—	Out of range uses replace mode. Modes 2, 3 and 4 based on line or fill pattern mask
	intin(0) = 1, replace = 2, transparent (mask 1s) = 3, XOR = 4, reverse transparent (mask 0s)							
Set a colour `vs_color`	14	0	0	4	0		—	Redefine a colour
	intin(0) = colour index (2) = red (4) = green (6) = blue } colour intensity 0 to 1000					In mono any colour is set to white	•	No action if a lookup table is not available or 'out of range'.
Set polyline line type `vsl_type`	15	0	0	1	1 *		—	All devices support at least six line styles.
	intin(0) = line style 1 = solid, 2 = long dash, 3 = dot, 4 = dash–dot, 5 = dash, 6 = dash–dot–dot, 7 = user defined. *(see below)*							User defined defaults to solid until defined.
Set user defined polyline `vsl_udsty`	113	0	0	1	0		—	User defined pattern for line, MSB is first pixel.
	intin(0) = line pattern word (16 bits)							
Set polyline width `vsl_width`	16	1	1	0	0		—	On error width is set nearest below Use odd numbs >= three.
	ptsin(0) = line width (2) = zero					ptsout(0) = width (2) = zero		
Set poly–line colour `vsl_color`	17	0	0	1	1 *		—	Set colour for polyline ops.
	intin(0) = colour index							

* denotes intout() is actual value of intin() used.

Attribute functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Set polyline end style vsl_ends	108	0	0	2	0		—	0 = square (def't) 1 = arrow 2 = rounded intin(0) = start style (2) = end style
Set polymarker type vsm_type	18	0	0	1	1 *		—	All devices support at least six markers. intin(0) = marker type 1 = dot 2 = plus 3 = asterisk 4 = square 5 = cross 6 = diamond Defaults on error to asterick
Set polymarker height vsm_height	19	1	1	0	0		—	Height set is nearest below. ptsin(0) = zero (2) = y-axis ht ptsout(0) = x–axis width (2) = y-axis height
Set poly–marker colour vsm_color	20	0	0	1	1 *		—	Set colour for polymarker ops. intin(0) = colour index
Set character height vst_heig	12	1	2	0	0		—	Size is of character. ptsin(0) = zero (2) = height ptsout(0) = char width (2) = char height (4) = cell width (6) = cell height V char
Set character cell height vst_point	107	0	2	1	1 *		—	Size is of cell. intin(0) = cell point size (1/72) ptsout(0) = char width ptsout(2) = char height ptsout(4) = cell width ptsout(6) = cell height V cell
Set char baseline vector vst_rotation	13	0	0	1	1 *		—	Angular range 0 to 3600 intin(0) = angle requested

Attribute functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Set text face vst_font	21	0	0	1	1 *		—	Face 1 is built in (System face) intin(0) = face selection
Set graph text colour vst_color	22	0	0	1	1 *		—	Set colour for next text. default 1 intin(0) = text col. index
Set text special effect vst_effects	106	0	0	1	1 *		—	Default to standard text Effect on if bit = 1 intin(0):bits 0 to 5 set effects Thick,light,skew,underline, outline,shadow
Set graphic text position vst_alignment	39	0	0	2	2 *		—	Left/right/centre justify. Vertical position defaults to base (= 0) intin(0) = 0,lft 1,cntr 2,rt (2) = 3,4,0,1,2,5 respectively bottom,descent,base,half,ascent,top
Set fill interior style vsf_interior	23	0	0	1	1 *		—	Set future polygon fill style. intin(0) = 0 to 4 respectively 0_hollow,1_solid,2_pattern,3_pattern,4_user defined
Set fill style index vsf_style	24	0	0	1	1 *		—	Set pattern or hatch type. No effect if interior hollow, solid or usrdef intin(0) = 0,n where n = solid colour 2,1 to 24 patterns 3,1 to 12 hatch
Set fill colour index vsf_color	25	0	0	1	1 *		—	Set future polygon fill colour intin(0) = colour index

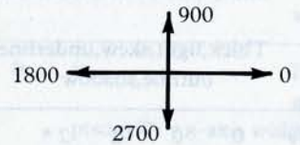
The set fill style index can be found with the colour command in the ST BASIC sourcebook.

Attribute functions – continued

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Set fill peri visible `vsf_perimeter`	104	0	0	1	1 *		—	Set on/off fill outline
	intin(0),0_invisible, <>0_visible							
Set user –defined fill pattern `vsf_udpat`	112	0	0	16 × n	0		—	Pattern 16 words/plane Bit 15 word one upper left bit.
	intin(0–15) = 1st plane (16–31) = 2nd plane etc.							

* denotes intout() is actual value of intin() used.

Notation used for angular specifications



The set fill style index can be found with the colour command in the ST BASIC sourcebook.

RASTER OPERATIONS

Raster operations are the manipuation of rectangular blocks of bits in memory or pixels on screen, the area is defined in memory form definition blocks (MFDB) that consists of:

Offset		
$0	Memory pointer	32–bit address of pixel 0,0
$4	Width pixels	Raster area dimensions
$6	height pixels	
$8	Word width	Pixel width/word size
$A	Format flag	1 = standard, 0 = device specific
$C	Memory planes	No. of planes in raster area
$E	Reserved	3 reserved words

The raster planes word–bit–pixel relationship follows the format shown in the TOS overview (Chapter 2), the top left hand corner pixel address being 0,0.

The colour index tables take a non–standard form and care should be taken to ensure correct colour usage.

Pixel	*Index*	*Colour*	*Pixel*	*Index*	*Colour*
0000	0	white	1000	9	grey
0001	2	red	1001	10	light red
0010	3	green	1010	11	light green
0011	6	yellow	1011	14	light yellow
0100	4	blue	1100	12	light blue
0101	7	magenta	1101	15	light magenta
0110	5	cyan	1110	13	light cyan
0111	8	low white	1111	1	black

Raster operations perform logical translations of the source to the destination over the original destination pixel area. The required logic operation is passed as an argument in intin(0) as follows:

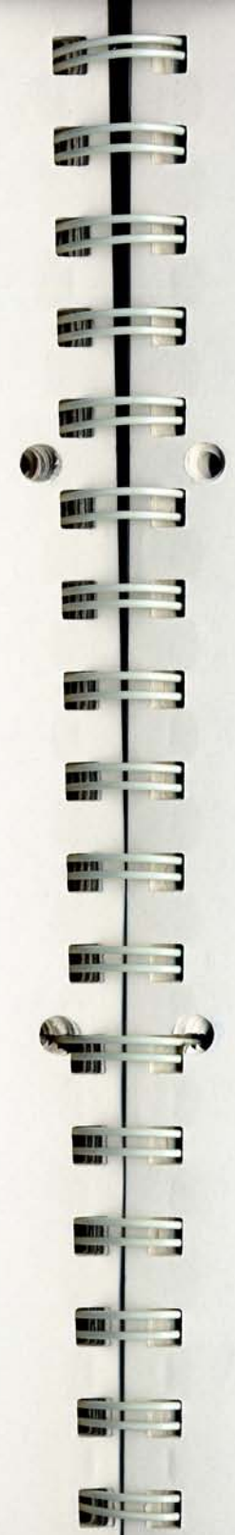
Mode	Function	Mode	Function
0	D' = 0 (all white)	8	D' = NOT [S OR D]
1	D' = S AND D	9	D' = NOT [S XOR D]
2	D' = S AND [NOT D]	10	D' = NOT D
3	D' = S	11	D' = S OR [NOT D]
4	D' = [NOT S] AND D	12	D' = NOT S
5	D' = D	13	D' = [NOT S] OR D
6	D' = S XOR D	14	D' = NOT [S AND D]
7	D' = S OR D	15	D' = 1 (all black)

S = Source
D = Destination
D' = Destination pixel final state

Mode 3 = replace
Mode 4 = erase
Mode 6 = XOR

Raster operations

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Copy raster opaque vro_cpyfm	109	4	0	1	0		—	Copy rect block from source to destination. If source <> dest'n then source size used. ptsin a,b,c, d,e,f,g,h
	intin(0) = logic op							
	cntrl $ E = Address.L of source MFDB							
	cntrl $12 = Address.L of destination MFDB							
Copy raster trans-parent vrt_cpyfm	121	4	0	3	0		—	Copy mono block from source to colour dest'n If src <> dest'n then source size used. ptsin a,b,c, d,e,f,g,h
	intin(0) = write mode (2) = colour for 1s (4) = colour for 0s							
	cntrl $ E = Address.L of source MFDB							
	cntrl $12 = Address.L of destination MFDB							
	write mode	a replace, ink (2), paper (4) b transparent. If src = 0 then dest ink(2) unused c XOR mode. XOR source to planes (2)&(4)unused d rev tran. If src = 1 then dest paper(4) unused						



Raster operations – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Transfrm form vr_trnfm	110	0	0	0	0		—	Toggle raster area from standard to device–specific form.
	cntrl $ E = Address.L of source MFDB							
	cntrl $12 = Address.L of destination MFDB							(This block must be verified)
Get pixel v_get_pixel	105	1	0	0	2		—	Return pixel value and colour index
	ptsin(0) = x coor (2) = y coor				intout(0) = pixel value (2) = colour index			

INPUT FUNCTIONS

There are two types of input function generally provided by GEM:

Request and wait for reply, and

Request and sample current status.

Only the following are implemented on the Atari ST.

Input functions

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Set mouse form vsc_form	111	0	0	37	0		—	Redefine cursor pattern. Bit 15 of word 1 upper left bit of pattern Data under mask is saved.
	intin(0) = x coor			intin(6) = mask colour				
	(2) = y coor			(8) = data colour				
	(4) = 1, reserved				(usually 1)			
	($ A–$28) = 16 word cursor mask bits							
	($2A–$48) = 16 word cursor data bits							
Exchange timer int vector vex_timv	118	0	0	0	1		—	Goto user-written interrupt routine on timer tick. Int's disabled
	cntrl	$ E = Addr.L of new routine						
	ret'n	$12 = Addr.L of old routine						
	intout(0) = milliseconds per tick							
Show cursor v_show_c	122	0	0	1	0		—	
	intin(0) = 0, show cursor							
	<>0, show if no. of show calls = no. hide calls							
Hide cur v_hide_c	123	0	0	0	0		—	Hide cursor (def't)
	Operates as per 'show cursor'							
Sample mouse button state vq_mouse	124	0	1	0	1		—	Ret'n button state Left button LSB 1 = button pressed 0 = button up
	ptsout(0) = x coor							
	(2) = y coor							
	intout(0) = return button state							
Exchange button change vector vex_butv	125	0	0	0	0		—	Goto routine on button state ch'g Uses D0.W for button keys as above. Int's disabled
	cntrl	$ E = Addr.L user routine						
	ret'n	$12 = Addr.L old routine						
	(Save and restore registers)							

Input functions

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Exchange mouse movement vector vex_motv	126	0	0	0	0		—	Goto routine on mouse movement. D0.W & D1.W store x & y. Int's disabled
	cntrl	$ E = Addr.L user routine						
	retn	$12 = Addr.L old routine						
	x and y coordinates may be changed after being stored in hardware register							
Exchange cursor change vector vex_curv	127	0	0	0	0		—	Goto routine on cursor state chg D0.W & D1.W store x & y. Int's disabled
	cntrl	$ E = Addr.L user routine						
	ret'n	$12 = Addr.L old routine						
	routine can be used to draw special cursor.							
Sample keyboard state information vq_key_s	128	0	0	0	1		—	Return state of keyboard shift–alt - control keys 0 bit key up 1 bit key dn
	intout(0),	bit0, right shift						
		bit 1, left shift						
		bit 2, Control						
		bit 3, Alternate						

Functions calling user written code should not enable interrupts. Registers may need to be restored.

The following GEM VDI functions *are not* implemented on the Atari ST, brief details are included for completeness:

Input functions (Not implemented)

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Set input mode vsin_mode	33	0	0	2	1		—	Set i/p mode for device to request or sample. Intout = mode selected
	intin(0) = Logical i/p device							
	1 = locator, 2 = valuator							
	3 = choice, 4 = string							
	intin(2) = i/p mode (1 = req, 2 = sample)							

Input functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Input locator, request mode vrq_locator	28	1	1	0	1			Ret position of locator device. Screen tracks cursor until terminated by key/button press ptsin(0) = init x coor (2) = init y coor intout(0) = terminate character (LSByte) ptsout(0) = final x coor (2) = final y coor
Input locator, sample mode (cursor change event) vsm_locator	28	1	1/0	0	0/1		—	Ret position of locator device ptsin(0) = init x coor (2) = init y coor intout(0) = terminate character (LSByte) ptsout(0) = new x coor (2) = new y coor (A tablet or a mouse terminate chars begin at 20H, 32dec) If 2 locators, either may input
Input valuator, req mode vrq_valuator	29	0	0	1	2		—	Return value of valuator device - arrow keys, range 1 to 100. intin (0) = init value intout (0) = o/p value (2) = terminator
Input valuator, sample mode vsm_valuator	29	0	0	1	0/2		—	Return value of device. intin (0) = init value intout (0) = new value (2) = keypress, if event occurred
Input choice, req mode vrq_choice	30	0	0	1	1		—	Ret choice status of device chosen. If invalid return choice number intin(0) = Init choice number (range 1 to device dependant maximum)
Input choice, sample mode vsm_choice	30	0	0	0	1		—	Ret choice status of device chosen. intout(0) = 0 if unsuccessful. intout(0) = 0, choice number cntrl $8 = 0, nothing occured = 1, sampled ok

If	*o/p's*	
new coor	1	0
key press	0	1
no i/p	0	0
key press & new coor	1	1

cntrl $8	*event*
0	none
1	val ch'ge
2	keypress

Input functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Input string, request mode vrq_string	31	1	0	2	L		—	Ret a string from specified device. Terminate on CR or intout full. If intin(0) is neg keyboard def D.5 ptsin(0) = scr x coor (2) = scr y coor intin(0) = max string length (2) = 0, no echo 1, echo at ptsin L = array length
Input string, sample mode vsm_string	31	1	0	2	0/>0		—	Ret a string from specified device. Terminate on CR, intout full or no more data. If intin(0) is neg pg D.5 def kybd. ptsin(0) = scr x coor (2) = scr y coor intin(0) = max string len (abs) (2) = 0, no echo 1, echo at ptsin cntrl(8) = 0, no chars returned

INQUIRE FUNCTIONS

The inquire functions return the current attribute settings of a specific device.

Inquire functions

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Extended inquire function	102	0	6	1	45		—	Ret extra device info not in the open workstation call or return open workstation values.
		intin(0)	= 0, open workstation values					
			= 1, extended inquire					
		intout(0)	= 0, not screen					
			= 1, separate			scrn	Alpha	
			= 2, common				and	
			= 3, separate			image	graphics	
			= 4, common			memo		
						ry	controller	may not be the same as $4E (78)
		(2) = # pallette background colours						
		(4) = Text effects supported(op 106)						
		(6) = Scaling 0 = no, 1 = yes						
		(8) = Number of planes						
		($ A) = Support look up table: 0 = yes, 1 = no						
		($ C) = $ 16 × 16 pixel raster ops/s (speed factor)						
		($ E) = contour fill capability						
		($10) = Char rotate 0 = no, 1 = 90° steps only, 2 = continuous						
		($12) = $ writing modes available						
		($14) = Input mode 0 = none, 1 = request, 2 = sample						
		($16) = Text alignment 0 = no, 1 = yes						
		($18) = Inking ability 0 = no, 1 = yes						
		($1A) = Rubberbanding 0 = no, 1 = lines, 2 = lines & rects						
		($1C) = Maximum $ pts, –1 = no max						
		($1E) = Maximum intin, –1 = no max						
		($20) = Number of keys on mouse						
		($22) = Styles for wide lines: 0 = no, 1 = yes						
		($24) = Writing modes for wide lines						
		($26–58) = reserved, contain zero words						
vq_extnd		ptsout(0–$16) = reserved, contain zero words						

Inquire functions

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire colour representation	26	0	0	2	4		—	Ret value of colour index in RGB units. Intout(0) = –1 out of range
		intin(0) = req colour index						
		(2) = 0,ret colour val reqst						
		(0) = 1,ret col val available						
		intout(0) = colour index						
		(2) = red intensity(0–1000)						
		(4) = green intensity(0–1000)						
vq_color		(6) = blue intensity(0–1000)						
Inquire current polyline attributes	35	0	1	0	5		—	Ret all attribs that affect polylines
		ptsout(0) = line width						
		(2) = zero						
		intout(0) = line type						
		(2) = line colour						
		(4) = write mode						
		intout(6) = Start end style						
		(8) = Finish end style						
vql_attributes								
Inquire current polymarker attributes	36	0	1	0	3		—	Ret all attribs that affect polymarkers
		ptsout(0) = width						
		(2) = height						
		intout(0) = marker type						
		(2) = marker colour						
		(4) = writing mode						
vqm_attributes								
Inquire current fill area attributes	37	0	0	0	5		—	Ret all attribs that affect fill areas. f = VDI function
		intout(0) = interior style(f_23)						
		(2) = colour						
		(4) = fill style(f_24)						
		(6) = writing mode						
		(8) = fill perimeter status						
vqf_attributes								
Inquire current graphic text attributes	38	0	2	0	6		—	Ret all attribs that affect graphic text.
		ptsout(0) = char width						
		(2) = char height						
		(4) = cell width						
		(6) = cell height						
		intout(0) = current graphic text face						
		(2) = current graphic text colour (font)						
		(4) = baseline angular rotation (0–3600)						
		(6) = horizontal alignment						
		(8) = vertical alignment						
vqt_attributes		($A) = writing mode (f_32)						

Inquire functions – continued

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire text extent vqt_extent	116	0	4	n	0		—	Ret a rectangle that encloses specified string.

intin = no. words in text
ptsout (0) = x coor \ bottom
(2) = y coor left
(4) = x coor bottom
(6) = y coor right
(8) = x coor top
($A) = y coor right
($C) = x coor top
($E) = y coor left

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire character cell width vqt_width	117	0	3	1	1		—	Ret character cell width of spec'd character in current text face

intin(0) = char val in ADE form
ptsout(0) = cell width ptsout(2) = 0
(4) = left char delta (6) = 0
(8) = right char delta ($A) = 0
intout(0) = ADE of inquire value
–1 if invalid character

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire face name and index vqt_name	130	0	0	1	33		—	Return 32 char face descriptor (text).

intin(0) = Font id (1 is base)
intout(0) = ID (Vst_font i/e)
(2–$40) = 32 ADE codes 1st 16 chars face
2nd 16 style and weight

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire cell array vq_cellarray	27	2	0	0	n		—	Return cell array definition of pixels.

cntrl $ E = row length colour index
$10 = #. rows array
return $12 = #. elems/row Used in colour
return $14 = #. rows Index array
return $16 0 = errors, 0 = none, 1 = pixel colour indeterminate
ptsin(0) = x coor ptsin(2) = y coor (lower left)
(4) = x coor (6) = y coor (upper right)
intout(0) = colour index array (1 row at a time)
–1 indicates indeterminate pixel colour

Inquire functions – continued

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire input mode vqin_mode	115	0	0	1	1		—	Ret'n current i/p mode for device.

intin(0) = logical dev. 1 = locator
2 = valuator
3 = choice
4 = string
intout(0) = i/p mode 1 = request
2 = sample

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire current face size information vqt_fontinfo	131	0	5	0	2		—	Ret'n current face size information.

ptsout(0) = max normal cell width
(2) = baseline to bottom
(4) = max extra skew width
(6) = baseline to descent line
(8) = left skew extra
($A) = baseline to half distance
($C) = right skew extra
($E) = baseline to ascent ADE = ASCII decimal equivalent
($10) = zero
($12) = baseline to top distance
intout(0) = 1st char in face
(2) = last char ADE

ESCAPE FUNCTIONS

The escape functions allow the programmer to access special device functions.

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Escape general format	5	—	—	—	—	id	—	
Inquire addressable alpha char cells `vq_chcells`	5	0	0	0	2	1	—	Get # of vertical rows & horizontal columns for alpha cursor.
		intout(0) = #rows (2) = #columns –1 no cursor addressing						
Exit alpha mode `v_exit_cur`	5	0	0	0	0	2	—	Enter graphics mode and exit alphameric mode.
Enter alpha mode `v_enter_cur`	5	0	0	0	0	3	—	Exit graphics mode and enter alphameric mode.
		cursor set to upper left of char cell						
Alpha cursor up `v_curup`	5	0	0	0	0	4	—	Move alpha cursor up one row.
		Do nothing if at top						
Alpha cursor down `v_curdown`	5	0	0	0	0	5	—	Move alpha cursor down one row
		Do nothing if at bottom						
Alpha cursor right `v_curright`	5	0	0	0	0	6	—	Move alpha cursor right one column.
		Do nothing if right edge						
Alpha cursor left `v_curleft`	5	0	0	0	0	7	—	Move alpha cursor left one column.
		Do nothing if at left edge						

Escape functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Home alpha cursor `v_curhome`	5	0	0	0	0	8	—	Move cursor to home position.
		Home, usually top left						
Erase to end of alpha screen `v_eeos`	5	0	0	0	0	9	—	Erase from current cursor position to end of screen.
		No cursor position change						
Erase to end of alpha text line `v_eeol`	5	0	0	0	0	10	—	Erase from current cursor position to end of line.
		No cursor position change						
Direct alpha cursor address `vs_curaddress`	5	0	0	2	0	11	—	Place cursor at specified row & column.
		intin(0) = row (1 to n) (2) = column (1 to n)						
Output cursor addressable alpha text `v_curtext`	5	0	0	n	0	12	—	Display a string of alpha text from current cursor position.
		n = # chars in string intin() = text string in ADE						
Reverse video on `v_rvon`	5	0	0	0	0	13	—	Display following text in reverse.
Reverse video off `v_rvoff`	5	0	0	0	0	14	—	Display following text in normal video
Inquire current alpha cursor address `vq_curaddress`	5	0	0	0	2	15	—	Return current alpha cursor position.
		intout(0) = row# (minimum one) (2) = column# (minimum one)						

Escape functions – continued

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Inquire tablet status vq_tabstatus	5	0	0	0	1	16	—	Return availab'ty status of tablet, mouse, j'stk etc.
	intout(0) = 0, not available 1, available							
Hard copy v_hardcopy	5	0	0	0	0	17	—	Copy screen to specific printer.
	may not be implemented							
Place graphic cursor at location v_dspcur	5	2	0	0	0	18	—	Place crosshair on screen.
	ptsin(0) = x coor (2) = y coor							
Remove last graphic cursor v_rmcur	5	0	0	0	0	19	—	

The following Escape functions are *not* implemented on the Atari ST, but are included for completeness; as is a discussion on VDI bit image file format.

Escape functions (Not implemented)

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Form advance v_form_adv	5	0	0	0	0	20	—	Pages printer but keeps screen display
Output window v_output_window	5	2	0	0	0	21	—	Copies specified window to printer Adjacent pictures may not join.
	ptsin(0) = x coordinate (2) = y coordinate } window corner (4) = x coordinate (6) = y coordinate } opposite corner							
Clear display list v_clear_disp_lis	5	0	0	0	0	22	—	Clear screen list without paging printer
Output bit image v_bit_image	5	0–2	0	L+2	0	23	—	Enables printer to process bit image file. by specifying or by default
	cntrl(2) = 0, get coordinates from file = 1, upper left specified							
	ptsin(0) = x upper left (2) = y upper left (4) = x lower right (6) = y lower right } coordinates if specified							
	intin(0) = Aspect ratio flag 0 = ignore, 1 = pxl ratio, 2 = page ratio (2) = Scaling 0 = uniform, 1 = x and y circles (4) = First char file name (length L) (2n + 2) = Last (nth) char file name							Pixel ratio provides for printing
Select pallette vs_palette	5	0	0	1	1	60	—	Allows IBM compatable pallette selection.
	intin(0) = 0, use red, green, brown = 1, use cyan, magenta, white intout(0) = pallette selected							
Inquire pallette film types vqp_films	5	0	0	0	125	91	—	Return film drier descriptor string
	intout 5 sets 25 ADE byte strings							

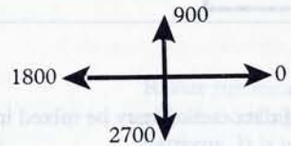
Escape functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Inquire pallette driver state vqp_state	5	0	0	0	20	92	—	Return film driver status block. intout(0) = port # 0 = 1st comms (2) = film number (0 to s) (4) = lightness control(-3 + 3) 1/3 f_stop step (6) = 0 noninterlace, 1 = interlace (8) = planes(1 to 4) (ADE format) ($0A-$28) = 2 char colour code for 8 colour indexes
Set pallette driver state vsp_state	5	0	0	0	20	93	—	Set film driver status block. intout(0) = port no. 0 = 1st comms (2) = film number (0 to 4) (4) = lightness cntrl(-3 + 3) 1/3 f stop step (6) = 0 noninterlace, 1 = interlace ($8-$26) = colour codes for 16 colours
Save pallette driver state vsp_save	5	0	0	0	0	94	—	Save current driver state to disk
Supress pallette messages vsp_message	5	0	0	0	0	95	—	Supress user prompts and error messages
Pallette error inquire vqp_error	5	0	0	0	1	96	—	Return error code intout(0) = 0, no error = 1, open dark slide for print film = 2, no port at specified location = 3,pallette not found at port specified = 4,video cable disconnected = 5,OS does not allow memory allocation = 6,not enough memory for buffer = 7,memory not deallocated = 8,driver file not found = 9,driver file incorrect type = 10,prompt user to process print film

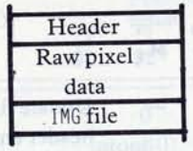
Escape functions – continued

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	GDP $A	Device name $C	Comments
Update metafile extents v_meta_extents	5	2	0	0	0	98	—	Update file header enabling application to get indication of a min window. ptsin(0) = min x (2) = min y (4) = max x (6) = max y bounding rectangle
Write metafile item v_write_meta	5	n	0	1	0	99	—	Intin and ptsin data written to metafile with a sub opcode >100 ptsin user defined data intin user defined data intin(0) = sub-opcode Sub opcodes 0 to 100 reserved
Change GEM VDI filename vm_filename	5	0	0	1	0	100	—	Rename metafile from GEMFILE.GEM to ____.GEM intin()path/filename up to 74 characters

Notation used for angular specifications



Bit image file format



There are two parts to the bit image file, a 16 word header, and a block of codified raw data.

File header

$ 0		upper left x	Bit image
$ 2		upper left y	
$ 4		lower right x	
$ 6		lower right y	
$ 8		page width	Source device
$ A		page height	
$ C		pixel width in microns	
$ E		pixel height in microns	
$10		bits per pixel	
$12 to $20		zero, reserved	

Raw data formats

The four methods of data coding may be mixed in any desired combination within a file.

Run length encoding (default)

$0	Run length	<128 bytes
$1	colour index	<256
	data	

Use a two byte subheader to define the data, which must be less than 128 bytes. The pixels may line wrap.

Extended run length encoding

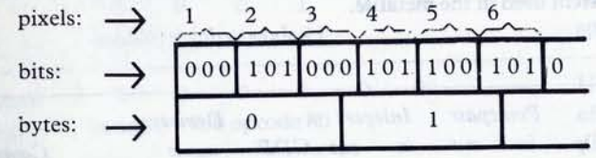
$0	Op code	–1
$1	Extended run length	<128 bytes
$2	colour index data	<256

To cater for pixel runs >127, the extended run includes a count of 128 pixels providing a range of 128 to 255 pixels. The pixel line may wrap.

Raster encoding

$0	Op code	–2
$1	#pix strm	
$2	packed colour indices	

Raster encoding packs colour indices into bytes in the following format:



Use either:

1 (black and white)
3 (four colour)
or 4 (sixteen colour)

bits per pixel format (offset $10 in the header).

Raster run encoding

$0	Op code	–3
$1	repeat count	<256
$2	# pix strm *	
$3	Packed colour indices	

Raster run encoding permits the efficient coding of repeated pixel patterns. It is in the same form as raster encoding but includes a repeat count in the header.

* #pix strm = the number of pixels in the stream

Metafile Sub opcodes

The Metafile functions *are not* implemented on the Atari ST, but are included for completeness of the GEM operating environment.

Output page (Not implemented)

There are two reserved GEM output codes for configuring the output page:

Physical page size, which defines the output area and coordinate window, specifying the coordinate system used in the metafile.

Output page

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Physical page size	5	0	0	3	0	99	—	Sub opcode 0
	intin(0) = sub opcode 0							
		(2) = page width				tenths of		
		(4) = page height				millimeters		
Coordinate window	5	0	0	5	0	99	—	Sub opcode 1
	intin(0) = sub opcode 1							
		(2) = x coor				lower left corner		
		(4) = y coor				of window		
		(6) = x coor				upper right corner		
		(8) = y coor				of window		

GEM Draw

There are a number of reserved GEM output codes used by GEM draw:

Group: Start and end enclose a set of primitives.

Draw area type primitive: Start and end indicate that enclosed functions are subject to the area type primitive block that follows the start function.

Attribute shadow: On and off indicate enclosed primitives are ignored as they are used to draw a drop shadow for the first primitive following 'off'.

Set no line style: Subsequent area type primitives are not outlined.

GEM Output codes

Function	*Op* $0	*Pointpair in* $2	*Pointpair out* $4	*Integers in* $6	*Integers out* $8	*GDP* $A	*Device name* $C	*Comments*
Start group	5	0	0	1	0	99	—	Bracket a set as a group
	intin(0) = sub opcode 10 of primitives							
End group	5	0	0	1	0	99	—	for a GEM DRAW application.
	intin(0) = sub opcode 11							
Start draw area type primitive	5	0	0	1	0	99	—	Use the vertices of the first primitive (except text)
	intin(0) = sub opcode 80							
End draw area type primitive	5	0	0	1	0	99	—	to define a GEM DRAW area type primitive.
	intin(0) = sub opcode 81							
Set attribute shadow on	5	0	0	1	0	99	—	Only draw a drop shadow on the first primitive,
	intin(0) = sub opcode 50							
Set attribute shadow off	5	0	0	1	0	99	—	ignore remaining shadow primitives until next off sub—opcode.
	intin(0) = sub opcode 51							
Set no line style	5	0	0	1	0	99	—	Subsequent area type primitives not to be outlined
	intin(0) = sub opcode 49							

GEM AES function calls

A set of application environment services (AES) function calls are available to the programmer, they consist of routines that make extensive use of the VDI function calls, and a dispatcher that provides a limited multitasking capability. The GEM VDI calls generally manage graphic outputs to peripheral devices, screen, printer etc. whereas GEM AES calls usually handle graphics input. The AES calls are grouped into eleven libraries that provide a variety of facilities:

Application library: controls the access to the other AES libraries.

Event library: responds to user inputs from mouse, keyboard or elapsed time.

Menu library: text options.

Object library: data collections that describes a displayed object, eg a box, an icon.

Form library: a means of obtaining information by the use of a list of questions.

Graphics library: a set of routines for manipulating the outline of a rectangular box.

Scrap library: routines that allow the interchange of data between applications.

File selector: user selection of a file from a displayed directory or a file via a filename and path.

Window library: manages up to eight GEM AES windows.

Resource library: provides the interface between the application and its data and files.

Shell library: enables an application to invoke another application and to keep track of the calling command and tail.

Within GEM AES there is a limited multitasking environment created by the dispatcher; a routine that activates processes sequentially simulating a multitasking environment. The dispatcher maintains two process queues, the 'ready' for processing list and the 'not ready' list, where processes are typically waiting for a user input, an input from another process or a specified time delay. Each 'ready' process is allowed a predefined period of CPU time before being returned to the end of the 'ready' queue, the environment is saved, the queues updated and control passed to the next item in the 'ready' queue.

Access to the AES functions is through an extended BDOS call and the AES parameter block (six longword pointers to the tables; control, global array, input and output attributes and input and output addresses). The AES parameter block, control table and global array have the following formats:

AES PARAMETER BLOCK

Offset		
$0	Control table pointer	
$4	Global array	
$8	I/P attrib table pointer	int_in
$C	O/P attrib table pointer	int_out
$10	I/P address table pointer	addr_in
$14	O/P address table pointer	addr_out
$18		

CONTROL TABLE

Offset		
$0	Opcode	
$2	Length i/p coor table	Table length in words
$4	Length o/p coor table	
$6	Length i/p addr table	Table length in longwords
$8	Length o/p addr table	
$A		

Global array

Offset		
$0	version	GEM AES version identification word
$2	count	Max # concurrent applcs supported
$4	id	Unique application identifier
$6	private	Longword user data as required
$A	ptree	Pointer to resource load address tree, initially zero
$E	reserved	Zero :Address of memory allocations
$12	reserved	Zero :Memory length; screen colours
$16	reserved	Zero
$1A	reserved	Zero
$1E		

The minimum size of an input table is one word, which must contain zero if no parameters are being passed.

Typical AES application call

A typical sequence of calls for an application might be:

Initialize and free unused memory, set up GEM parameter blocks and tables (APPL_INI [10] must be called first).

Open (virtual) workstation and get the screen resolution.

Load a resource file, applicable to the current screen resolution and number of colours available, into memory.

Get the address of specific resource objects and store them in memory.

Get the address of the resource menu bar and call 'display menu'.

Find the size and location of window (WIND_GET), identify window as a desktop (handle = 0), get the windows width/height (get_field = 4) and draw icons.

Wait for a user action, a keystroke, mouse button click or movement, GEM AES message or a specified time delay as either individual occurences or combined events.

Select from the menu, normally by moving the mouse to the menu bar. The message buffer is updated automatically and the process waiting for the input is moved to the 'ready list' and progressed to the next stage.

Reserve and then box a space to hold the dialog which is tested for an exit. On exit, any highlighting should be deselected.

Further user selections, could entail keyboard entries, icon selection etc.

One of the first operations of an application is to create an active window, which may be sized, redrawn, updated and finally closed.

Note that VDI calls use device handles and AES windows handles – further confusion may arise in the use of file handles – they are all different, beware!!!!

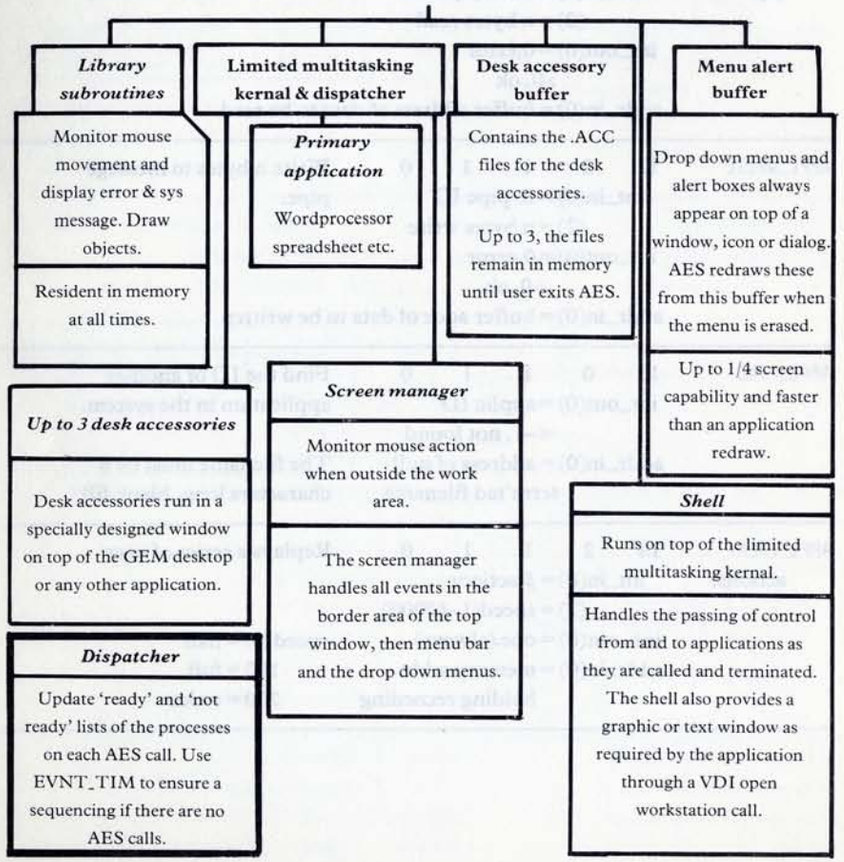
PARAMETER BLOCK SIZE

The numbers of parameters required by the various functions are detailed in the tabular format:

		Integers		*Addresses*		
Function	*Op*	*in*	*out*	*in*	*out*	*Comments*
	$0	*$2*	*$4*	*$6*	*$8*	

The table contains details of the parameter inputs and outputs; note that a zero indicates a block filled with a zero.

GEM AES components



APPLICATION LIBRARY

The application library functions initialize memory and data structures, terminate processes, communicate with other processes and record/replay user actions.

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
APPL_INIT	10	0	1	0	0	Initialize application and generate data structures prior to other AES function calls.
	intout(0) = application ID –1 failure, > = 0 ok put in global array					
APPL_READ pipe.	11	2	1	1	0	Read n bytes from message
	int_in(0) = from pipe ID (2) = n bytes read int_out(0) = 0,error >0, ok addr_in(0) = buffer address of data to be read					
APPL_WRITE	12	2	1	1	0	Write n bytes to message pipe.
	int_in(0) = to pipe ID (2) = n bytes write int_out(0) = 0,error >0, ok addr_in(0) = buffer addr of data to be written					
APPL_FIND	13	0	1	1	0	Find the ID of another application in the system.
	int_out(0) = applic ID = –1, not found addr_in(0) = address of null term'ted filename					The filename must be 8 characters long, blank fill.
APPL_TPLAY actions.	14	2	1	1	0	Replays a series of user
	int_in(0) = # actions (2) = speed(1–10000) int_out(0) = one (always) addr_in(0) = memory addr holding recording					speed 50 = half 100 = full 200 = twice

Application library – continued

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
APPL_TRECORD	15	1	1	1	0	Record a series of actions
(6 byte record)	int_in(0) = # actions int_out(0) = # recorded addr_in(0) = addr in memory to store records					1st word: 0 = timer, 1 = button, 2 = mouse, 3 = kybd lo longword hi: elapsed time ms; 0 = up, 1 = dwn/# clks; x pixels/y pixels; char/kybd status
APPL_EXIT	19	0	1	0	0	Let application library clean up environment when applic finish making calls
	int_out(0) = 0,error >0, ok					

EVENT LIBRARY

The event library routines monitor multiple and individual user inputs providing efficient polling of the clock, keyboard, mouse and message pipes.

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
EVNT_KEY	20	0	1	0	0	Return standard keyboard code (Appendix D.)
		int_out(0) = keycode press				
EVNT_BUTTON	21	3	5	0	0	Return mouse status on
		int_in(0) = wait #clicks				
		(2) = buttmask				
		(4) = button state				
		int_out(0) = No. clicks > = 1				
		(2) = x coor } on				
		(4) = y coor } event				
		(6) = button state				Button state bits 0 = up, 1 = down
		(8) = keystate				
EVNT_MOUSE	22	5	5	0	0	Return mouse status on leaving specified area
		int_in(0) return flag				Return flag 1, on area exit 0, on area entry
		(2) = x coor } area				
		(4) = y coor } position				
		(6) = width } pixel				
		(8) = height } coor				
		int_out(0) = Reserved (= 1)				
		(2) = x coor } on				
		(4) = y coor } event				
		(6) = button state				Button state bits 0 = up, 1 = down
		(8) = keystate				

Mask	*buttmask*	*Keystate*
Ox0001	butt left	right_shift
Ox0002	2nd butt	left_shift
Ox0004	3rd butt	control
Ox0008	up to 16	alternate

Mask	*buttmask*	*Keystate*
Ox0001	butt left	right_shift
Ox0002	2nd butt	left_shift
Ox0004	3rd butt	control
Ox0008	up to 16	alternate

Event library – continued

EVNT_MESAG 23 0 1 1 0 — Flag message, up to eight words, in message pipe.

int_out(0) = Reserved (= 1)

16 byte buffer { addr_in(0) = mess type ID →
(2) = ID of sender
(4) = 0 or length of message greater than 16 bytes

(6–14) = extra words

Addrin() extra word entries	
A = window handle	
B = x coor	G = Object index title
C = y coor	H = Object index item
D = width	I = menu item ID
E = height	(op 35 call return)
F = Page	J = top/left 0–1000
0 = up, 1 = down	Mouse
4 = lft, 5 = right	arrow
row 2 = up, 3 = down	click
col 6 = left, 7 = right	message

Messages entered FIFO, where message length > 16 byte use APPL_READ. Reading kills message.

ID	extra words required	Message function
10	GH	selected menu
20	ABCDE	redraw window
21	A	move work area to top
22	A	close window
23	A	toggle fullsize window
24	AF	scroll/page window
25	AJ	move window horizontal
26	AJ	move window vertica
27	ABCDE	resize window
28	ABCDE	move window
29	A	set new top window
40	I	desk_acc open message
41	I	desk_acc close message
50		ct_update
51		ct_move
52		ct_newtop

EVNT_TIMER 24 2 1 0 0 — Flag application that a specified length of time has past.

int_in(0) = lo longword
(2) = hi time ms
(0) = Reserved (= 1)

EVNT_MULTI 25 16 7 1 0 — Application waiting on one or more events

int_in(0) = Standard keyboard code
(2) = No. clicks
(4) = buttmask
(6) = button state — Button state 0_up, 1_down
(8) = flags } Mouse 1 area event
($A) = x coor
($C) = y coor
($E) = width
($10) = height
($12) = flags } Mouse 2 area event
($14) = x coor
($16) = y coor
($18) = width
($1A) = height
($1C) = low } longword time ms
($1E) = high

Mask	*buttmask*	*flags*
Ox0001	butt left	Keyboard
Ox0002	2nd butt	Button
Ox0004	3rd butt	Mouse 1
Ox0008	...	Mouse 2
Ox0010		Message
Ox0020		Timer

Flags show the type of event the application is waiting for, or occurred

continued...

Event library – continued

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
EVNT_MULTI *continued*						

int_out(0) = flag
(2) = x coor
(4) = y coor
(6) = button state
(8) = keystate
($A) = keycode press — Return std kybd code
($C) = # clicks> = 1 — # of button events/time
addr_in(0) = 16 byte buffer (see EVNT_MESAG op 23)

Keystate
Ox0001_right shift
Ox0002_left shift
Ox0004_control
Ox0008_alternate

EVNT_DCLICK	26	2	1	0	0	Get/set double click speed

int_in(0) = slow 0 to 4
(2) = 0_get, 1_set
int_out(0) = speed 1_new 0_old

Most applications will wait for a combination of events using the EVNT_MULTI call. When a required event occurs, the application will be moved from the 'not ready' list to the 'ready' list by the dispatcher, respond to the event and then return to the 'not ready' list to wait for the next event in the EVNT_MULTI sequence.

Be careful in using the right hand Atari mouse button, not all versions of GEM have two buttons, when considering portable software.

Keystroke selection

Some menu items support keystoke selection through the EVNT_MULTI call. On receipt of the specified key selection, the application should call MENU_TNORMAL to highlight the title to enable the user to see the selection actually made; deselect highlighting when the application has finished with the menu. The 16-bit keyboard event codes are given in Appendix D; use GRAF_MKSTATE to decode Control, Alternate and left and right SHIFT keys.

Icon selection

The bits for the required icon selection sequence are set by the application in the EVNT_MULTI call, button up or down state and a predefined number of clicks within a given space of time. On the event taking place, a bit value for the mouse and keyboard state is returned; the application needs to also call GRAF_MKSTATE to obtain the mouse's x and y coordinates and then make an OBJC_FIND call passing the x and y coordinates and the address of the window, desktop or application object tree containing its icons.

If OBJC_FIND reports the mouse covering an icon, its state should be changed to selected.

If the mouse does not cover an icon, the application should assume the user will select a group of icons by drawing an expanding rectangle around them. Call GRAF_MKSTATE to ensure the button is still depressed and then call GRAF_RUBBERBOX to provide the extent of the box when the button is released. The application should look for icons within the rectangle and change each icon from normal to selected via OBJC_CHANGE calls.

Menu library

The menu library routines provide the user with a textual menu choice from within an application, placing the mouse cursor over an enabled item and clicking the mouse button to make the selection.

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
MENU_BAR	30	1	1	1	0	Display/erase menu bar

int_in(0) = menu bar 0_erase, 1_draw
int_out(0) = error 0_yes, + ve_no
addr_in(0) = Object tree address that forms this menu

MENU_ICHECK	31	2	1	1	0	Display/erase menu item check mark

int_in(0) = menu item ID
(2) = 0_clear, 1_display (check mark)
int_out(0) = error 0_yes, + ve_no
addr_in(0) = Object tree address that forms this menu

MENU_IENABLE	32	2	1	1	0	Disable/enable menu item

int_in(0) = menu item ID
(2) = 0_disabled, 1_enabled (light/dark)
int_out(0) = error 0_yes, + ve_no
addr_in(0) = Object tree address that forms this menu

MENU_TNORMAL	33	2	1	1	0	Display menu title in reverse video

int_in(0) = menu item ID
(2) = 0_reverse, 1_normal video
int_out(0) = error 0_yes, + ve_no
addr_in(0) = Object tree address that forms this menu

Menu library – continued

Function	Op	Integers in	Integers out	Addresses in	Addresses out	Comments
	$0	$2	$4	$6	$8	
MENU_TEXT	34	1	1	2	0	Change menu text reverse video
	int_in(0) = menu item ID					
	int_out(0) = error 0_yes, + ve_no					
	addr_in(0) = Addr of new string for this item					
	(4) = Object tree addr for this menu					
MENU_REGISTER	35	1	1	1	0	Place desk accessory menu item string on desk menu and ret acc's menu ID
	int_in(0) = Desk accessory process ID					
	int_out(0) = menu item ID (0–5)					
	addr_in(0) = address of desk_acc menu text string.					

To display a menu bar, call the resource function RSRC_GADDR with the menu bar's (object) details to obtain the long address of the object tree root, call MENU_BAR with the address and set the routine to draw.

Menu bar control

The AES screen manager controls all user interaction with the menu bar in the following manner:

The user touches an item in the menu bar using the mouse cursor.

The screen manager receives a message that the cursor has entered the menu bar and enters the 'ready list'. It determines which item in the title bar the cursor touched, saves the screen under and displays the 'titles' menu; highlighting menu items as the cursor passes over them.

The application is held in the 'not ready' list while the screen manager has initiated open menus. When the user clicks the mouse on a menu item, the screen manager sends details of the object tree of the menu selected to the primary application's message buffer.

The dispatcher checks the 'not ready' list for the application process waiting for the message and moves it to the 'ready' list.

The EVNT_MULTI call returns a flag of the events that occurred, which may be read by the application and any action deemed appropriate by the application taken.

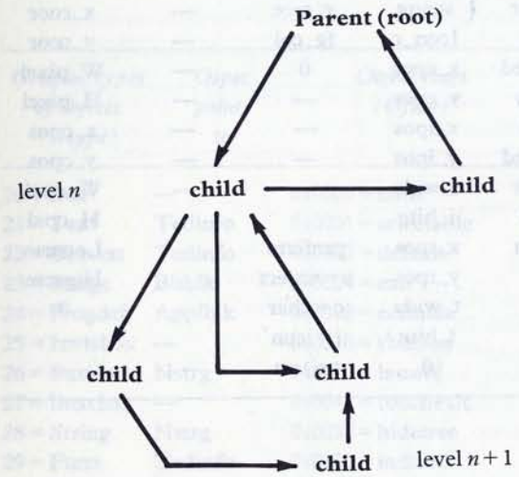
When the action is complete, the menu title is de-highlighted by the application making a MENU_TNORMAL call.

Object library

An object, described by a collection of data in a linked list (object tree), can be created, deleted, edited, drawn on the screen, and the object's position on the screen found, using the object library routines.

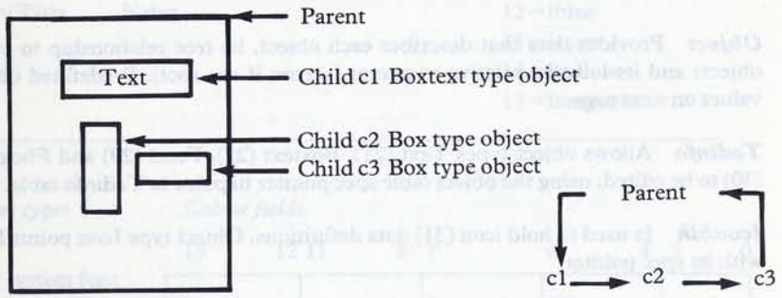
Object tree

An object consists of a parent and perhaps a number of different levels of children, who always reside within the parents display space. The tree is created by making seperate calls to the OBJC_ADD routine for each child or loaded from disk using RSRC_LOAD.



Each child points to a brother in a chain, if it has one. The last one points back to its parent.

Different objects may be created by only using parts of the tree.



The object library uses a number of additional tables, as well as the parameter block, control table and global arrays, to describe objects. The tables are accessed via the resource library routines and are as follows:

Additional object library word tables (bracketed items are longwords)

Offset	*Object*	*Tedinfo*	*Iconblk*	*Bitblk*	*Applblk*	*Parmblk*
($0)	Nextchild	Text	Mask	Image	Code	Tree
($2)	1stchild	string	string	pointer	pointer	pointer
($4)	Lastchild	Template	Data	W_arry	Loparm	Objindx
($6)	Otype	string	string	H_pixl	Hiparm	Oldst
($8)	Oflag	Vchar	Text	x_srce	—	Newst
($A)	OState	pointer	string	y_srce	—	x_coor
($C)	OSpec	Font	Icon_c	fg_col	—	y_coor
($D)		Reserved	x_cpos	0	—	W_pixel
($10)	x_coor	Justify	y_cpos	—	—	H_pixel
($12)	y_coor	Color	x_ipos	—	—	x_cpos
($14)	Width	Reserved	y_ipos	—	—	y_cpos
($16)	Height	Brdthk	i_wide	—	—	W_cpxl
($18)	—	txtlen	i_hite			H_cpxl
($1A)	—	tmplen	x_tpos	prefixes		Loparm
($1C)	(x&y		y_tpos	o = object		Hiparm
($1E)	rel to		t_wide	c = char		0
($20)	parent		t_hite	i = icon		—
($22)	or screen)	—	0	t = text		—

The tables, filled by the object library routines, are used in performing various functions:

Object Provides data that describes each object, its tree relationship to other objects and its location relative to parent (screen if the root). Predefined object values on next page.

Tedinfo Allows object types Text (21), Boxtext (22), Ftext (29) and Fboxtext (30) to be edited, using the object table spec pointer to point to Tedinfo table.

Iconblk Is used to hold icon (31) data definitions. Object type Icon points here with its spec pointer.

Bitblk Object type Image (23) uses this to draw bit images like cursors and icons.

Applblk Is used to locate and call an application defined routine that draws and or changes an object. The object type Progdef (24) spec pointer points here.

Parmblk Storage of data used by the application defined routine above (applblk) and pointed to by the code pointer.

Object libraries

Routines which edit, create and draw data describing objects that appear on the screen, boxes, characters, icons etc.

There are some predefined values for the table entries:

Graphic types of objects (Otype)	*Ospec points to*	*Object flags (Oflag)*	*Object colors (color)*
20 = Box	—	0x0000 = none	0 = white
21 = Text	Tedinfo	0x0001 = selectable	1 = black
22 = Boxtext	Tedinfo	0x0002 = default	2 = red
23 = Image	Bitblk	0x0004 = exit	3 = green
24 = Progdef	Applblk	0x0008 = editable	4 = blue
25 = Invisbox	—	0x0010 = rbutton	5 = cyan
26 = Button	Nstrg	0x0020 = lastobj	6 = yellow
27 = Boxchar	—	0x0040 = touchexit	7 = magenta
28 = String	Nstrg	0x0080 = hidetree	8 = white
29 = Ftext	Tedinfo	0x0100 = indirect	9 = black
30 = Fboxtext	Tedinfo		10 = lred
31 = Icon	Iconblk		11 = lgreen
32 = Title	Nstrg		12 = lblue
			13 = lcyan
			14 = lyellow
			15 = lmagenta

Font types

3 = system font
5 = small font

Colour fields

15	12 11	8 7	6	4 3	0
border colour	text colour	0_tran 1_repl	fill type	inside colour	

Object states
(Ostate)

0x0000	normal
0x0001	selected
0x0002	crossed
0x0004	checked
0x0008	disabled
0x0010	outlined
0x0020	shadowed

Ospec 32-bit word/byte values

	Low word	*High word*	
		Low byte	*High byte*
Box	colour	0	0
Invisbox	colour	brdthk	0
Boxchar	colour	0	char

Editable text

field definitions	*justification*
0 = edstart	(Justify)
1 = edinit	0 = left justified
2 = edchar	1 = right justified
3 = edend	2 = centered

Borderthickness
(brdthk)

0	none
1 to 128	inside
–1 to –127	outside (in pixels)

Allowable valid characters (Vchar pointer)

9	only digits 0 to 9
A	only uppercase A to Z and space
a	upper and lowercase A to Z and space
N	0 to 9, uppercase A to Z and space
n	0 to 9, upper and lowercase and space
F	all valid DOS filename chars, plus ?, *, :
P	all valid DOS pathname chars, plus \, :, ?, *
p	all valid DOS pathname chars, plus \, :
X	anything

Object libraries – continued

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
OBJC_ADD	40	2	1	1	0	Add an object to an object tree.
	int_in(0) = Parent ID					
	(2) = Child ID (item to add)					
	int_out(0) = error 0_yes, + ve_no					
	addr_in(0) = Object tree addr of parent and child					
OBJC_DELETE	41	1	1	1	0	Delete an object from an object tree
	int_in(0) = Object to delete					
	int_out(0) = error 0_yes, + ve_no					
	addr_in(0) = Object tree address with object in it					
OBJC_DRAW	42	6	1	1	0	Draw an object in an object tree
	int_in(0) = start object					
	(2) = draw 0_obj only, nth_level					
	(4) = x coor					
	(6) = y coor		Clip			
	(8) = width		rectangle			
	($A) = height					
	int_out(0) = error 0_yes, + ve_no					
	addr_in(0) = Object tree address with object in it					
OBJC_FIND	43	4	1	1	0	Find an object under the mouse form
	int_in(0) = search start object					
	(2) = levels of search					
	(4) = x coor		mouse			
	(6) = y coor		location			
	int_out(0) = –1_no obj, 0 to n = # of object in tree					
	addr_in(0) = Object tree address of search start object					
OBJC_OFFSET	44	1	3	1	0	Find obj. screen rel x/y coords
	int_in(0) = object to locate					
	int_out(0) = error 0_yes, + ve_no					
	(2) = x coor		relative			
	(4) = y coor		to screen			
	addr_in(0) = Object tree address with int_in(0) in it					

Object libraries – continued

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
OBJC_ORDER	45	2	1	1	0	Reorder an object within a list.
		int_in(0) = Object to be moved				
		(2) = new pos (0 = bot. level . . 1 . . to –1 top)				
		int_out(0) = error 0_yes, + ve_no				
		addr_in(0) = Object tree address with int_in(0) in it				
OBJC_EDIT	46	4	2	1	0	Edit object text.
		int_in(0) = text object to be edited				
		(2) = user input character				
		(4) = next character index in text string				
		(6) = 0,reserved				
		= 1,fmt str using text/template strings				
		= 2,validate against Tedinfo valid_char, update and display.				
		= 3,turn off text cursor				
		int_out(0) = error 0_yes, + ve_no				
		(2) = next char index after operation				
		addr_in(0) = Object tree addr of obj with text in it				
OBJC_CHANGE	47	8	1	1	0	Changes an object's state
		int_in(0) = object to be changed				
		(2) = zero, reserved				
		(4) = x coor				
		(6) = y coor clip				
		(8) = width rectangle				
		($A) = height				
		($C) = object state new value				
		($E) = redraw 0_no, 1_yes				
		int_out(0) = error 0_yes, + ve_no				
		addr_in(0) = Object tree address				

To display an icon, calculate the desktop windows work area using a WIND_GET call and use OBJC_DRAW to draw the icon in the work area. The icons position within the window is held by the 'Iconblk' structure.

Form library

A set of routines that enable the user to reply to a list of questions, either by checking off boxes or entering text.

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
FORM_DO	50	1	1	1	0	Monitor interaction with a form.
		int_in(0) = object number				
		int_out(0) = object # that caused the exit				
		addr_in(0) = object tree address				
FORM_DIAL	51	9	1	0	0	Reserve/Free dialog box screen area.
		int_in(0) = flag				
		(2) = x coor				
		(4) = y coor	small box			Flag:
		(6) = width				0 = res screen space for dialog box
		(8) = height				
		($A) = x coor				1 = expanding box
		($C) = y coor	large box			2 = shrinking box
		($E) = width				3 = free screen space
		($10) = height				
		int_out(0) = error 0_yes, 1_no				
FORM_ALERT	52	1	1	1	0	Display an alert.
		int_in(0) = exit button				0 = no default exit
		int_out(0) = chosen exit				1 = 1st exit button
		addr_in(0) = address alert string				2 = 2nd exit button..
FORM_ERROR	53	1	1	0	0	Display an error box.
		int_in(0) = DOS error code				
		int_out(0) = exit button code (as above)				
FORM_CENTER	54	0	5	1	0	Centre dialog box on screen
		int_out(0) = one, reserved				
		(2) = x coor				Of
		(4) = y coor				centered
		(6) = width				object
		(8) = height				tree
		addr_in(0) = dialog object tree address				

The forms library routines enable the user to respond to a typical printed style of form on the screen in a question and answer mode without tying up the applications resources. The forms library also provides a consistent application/user response format.

The forms have three optional types of user response, they are:

Check a single box
Check a combination of boxes
Provide a typed response

These may be used any number of times in any combination. Finally the user exits typically via an OK or CANCEL button.

Taking a dialog as an example:

To display a dialog, which will appear in the centre of the screen, call resource function RSRC_GADDR to get the address of the dialogs object tree. Call FORM_DIAL to reserve screen space and then call OBJC_DRAW to draw the dialog.

The application should call FORM_DO to monitor user interaction with the dialog box. Where user changes have been made, the application may use OBJC_CHANGE to reset initial values, in particular dehighlight selected buttons. It may also be necessary to save some changes made to dialogs.

To exit from the dialog, call FORM_DIAL to release the screen space, the application which should be in an EVNT_MULTI wait state can redraw the screen using an OBJC_DRAW call.

A nicer display may be achieved if FORM_DIAL is used to draw expanding and shrinking boxes on start up and finish of the dialog sequence.

Edit keys

Keys have certain specified meanings for editing the text fields of forms and dialog boxes:

←and →: Move left or right within the field.

↓and TAB: Move to first free space of the next field.

↑: Move to first free space of previous field.

DELETE: Delete character following cursor without moving cursor.

BACKSPACE Delete character to the left of the cursor, move cursor and following text one space left.

RETURN End edit and terminate if either OK or CANCEL type buttons are default objects otherwise ignore.

ESCAPE: Clear all characters from the field.

Alerts

Alerts, which are used by GEM AES to handle error conditions, contain one of four pictorial designs; nothing, note icon, wait icon and the stop icon, and upto a maximum of 5 lines of 40 character width text (each line being seperated by the | symbol) and up to 3 exit buttons, each containing up to 20 characters of text.

A special case alert is the error box which reports errors in DOS terminology (Appendix I).

A typical set of object structures for an alert on a mono screen box with some textual information and OK and CANCEL buttons might be:

Offset	*Object structure element*	*Box*		*HELP Text*	*OK Boxtext*	*CANCEL Boxtext*		*Comments* *Pntr to next obj*
0	nextchild	–1		2	3	0	←	–1 root
2	1stchild	1		–1	–1	–1	}	–1 lowest
4	Lstchild	3		–1	–1	–1	}	level
6	Otype	20		21	22	22		
8	Oflag	0		0	1_selectd	2_default		
$A	Ostate	0		0	0	0		
$C	Ospec	00020007L		0L	0L	0L		
$10	x-coor	90	Rel	86	374	374		
$12	y_coor	150	to	16	18	50		
$14	width	454	scrn	272	64	54		
$16	height	98		64	16	16		

↑ Relative to parent (Box)

Alerts – continued

Offset	*Tedinfo structure element*	*Box*	*HELP Text*	*OK Boxtext*	*CANCEL Boxtext*
0	Text string	—	help	ok	cancel
4	Tmpltstrg	—	0	0	0
8	Vcharpointer	—	0	0	0
$C	Font	—	3	3	3
$D	Reserved	—	0	0	0
$10	Justify	—	0_left	2_center	2_center
$12	colour	—	00020000L	00020000L	00020000L
$14	Reserved	—	0	0	0
$16	Brdthk	—	0	–2	–2
$18	txtlen	—	0	0	0
$1A	tmplen	—	0	0	0

The form library follows the tree from root to children in displaying the form object

Graphics library

The graphics library routines enable the programmer to manipulate the rectangular outline of a box.

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
GRAF_RUBBERBOX	70	4	3	0	0	Draw box expanding & contracting from a fixed point with mouse
		int_in(0) = x coor				of
		(2) = y coor				box
		(4) = min pixel width				
		(6) = min pixel height				
		int_out(0) = error 0_yes, + ve_no				
		(2) = width				when button
		(4) = height				last released
GRAF_DRAGBOX	71	8	3	0	0	Move a box and keep the mouse ptr at same position inside box.
		int_in(0) = width				of
		(2) = height				box
		(4) = x coor				being
		(6) = y coor				dragged — Height and width in pixels
		(8) = x coor				
		($A) = y coor				Boundary
		($C) = width				rectangle
		($E) = height				
		int_out(0) = error 0_yes, + ve_no				
		(2) = x coor				when button
		(4) = y coor				released
GRAF_MOVEBOX	72	6	1	0	0	Draw a moving box
		int_in(0) = width				
		(2) = height				
		(4) = x coor				Initial
		(6) = y coor				position — Height and width in pixels
		(8) = x coor				Final
		($A) = y coor				position
		int_out(0) = error 0_yes, + ve_no				

Graphics library – continued

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
GRAF_GROWBOX	73	8	1	0	0	Draw expanding box outline
		int_in(0) = x coor (2) = y coor (4) = width () = height		Initial position		
		()= x coor ($A) = y coor ($C) = width ($E) = height		Final position		Height and width in pixels
		int_out(0) = error 0_yes, + ve_no				
GRAF_SHR	74	8	1	0	0	Draw shrinking box outline
		int_in(0) = x coor (2) = y coor (4) = width (6) = height		Final position		
		(8) = x coor ($A) = y coor ($C) = width ($E) = height		Initial position		Height and width in pixels
		int_out(0) = error 0_yes, + ve_no				
GRAF_WAT	75	4	1	1	0	Track mouse pointer and button in and outside the box.
		int_in(0) = reserved (2) = object tree (4) = in the box (6) = out of box		obj. state		
		int_out(0) = 0_outside, 1_inside the box addr_in(0) = address object tree containing box				
GRAF_SLI	76	3	1	1	0	Keep sliding box inside parent box.
		int_in(0) = parent index (2) = object index (slider) (4) = motion 0_horiz, 1_vert				
		int_out(0) = 0_lft/top to 1000_right/bottom addr_in(0) = Address object tree containing slider & parent				

Graphics library – continued

Function	Op $0	Integers in $2	Integers out $4	Addresses in $6	Addresses out $8	Comments
GRAF_HANDLE	77	0	5	0	0	Ret GEM VDI handle for open screen workstation
		int_out(0) = VDI handle (2) = width (4) = height (6) = width (8) = height			char cell syst. font box for syst. font	
GRAF_MOUSE	78	1	1	1	0	Permit application to change predefined mouse.
		int_in(0) = 0 arrow = 1 text cursor (vertical bar) = 2 bee (hourglass-IBM GEM) = 3 hand with pointing finger = 4 flat hand, extended fingers = 5 thin cross hair = 6 thick cross hair = 7 outline cross hair = 255 mouse form stored in addr_in(0) = 256 hide mouse form = 257 show mouse form				
		int_out(0) = error 0_yes, + ve_no addr_in(0) = 35 wd buf: mouse form def'n block				(VDI op 111)
GRAF_MKSTATE	79	0	5	0	0	Return mouse location button and keyboard state.
		int_out(0) = 1, reserved (2) = x coor (4) = y coor (6) = Butstate (8) = keystate			mouse location 0_up 1_dn	

Mask	Butn	Key
Ox0001	lift	Rt SHIFT
Ox0002	2nd	Left SHIFT
Ox0004	3rd	CTRL
Ox0008		ALT

GEM AES provides the graphic routines to manipulate the rectangular outline of a box which are based on GEM VDI routines. Graphics applications should use GEM VDI directly for graphic output to avoid any loss in performance through the AES overhead.

Scrap library

The scrap library consists of routines that manage the interchange of information between applications. Data is either deleted or copied from the source to the clipboard (disk file named scrap), which only holds one document; and then pasted (copied) from the clipboard (disk) to the target application.

Function	*Op*	*Integers in*	*Integers out*	*Addresses in*	*Addresses out*	*Comments*
	$0	*$2*	*$4*	*$6*	*$8*	
SCRP_READ	80	0	1	1	0	Read the current scrap directory on the clipboard
	int_out(0)		= error 0 = yes + ve no			
	addr_in(0)		= buffer address into which scrap directory is copied.			
SCRP_WRITE	81	0	1	1	0	Write new scrap directory to clipboard. (Cut & Copy)
	int_out(0)		= error 0 = yes + ve no			
	addr_in(0)		= buffer address from which scrap directory is copied to clipboard.			

The scrap data is held on disk in a file named scrap, the extension identifies the type of data:

.TXT	ASCII text string
.DIF	Spreadsheet data
.GEM	Metafile – GEM VDI type graphic images
.IMG	Bit image – GEM VDI standard form

Applications access the data via GEM BDOS file system calls to:

Search
Create a file
Open a file
Read a file
Write a file
Close a file
Delete a file and
Get file size.

File selector library

The file selector library routine enables the programmer to select file from a displayed directory or to type in a filename and path.

Function	*Op*	*Integers in*	*Integers out*	*Addresses in*	*Addresses out*	*Comments*
	$0	*$2*	*$4*	*$6*	*$8*	
FSEL_INPUT	90	0	2	2	0	Display file selector box and monitor user interaction with it.
	int_out(0)		= error 0 = yes + ve no			
	(2)		= exit button 0_cancel 1_ok			
	addr_in(0)		= buffer address of intial directory specification (if not updated holds last dir spec user selected)			
	addr_in(4)		= buffer address of initial selection displayed in file selector dialog box (If not updated it holds last selection)			

This routine displays a file directory dialog box, the user either selects a filename directly from the directory list using a mouse or types in a filename to create a new file.

The file directory dialog box displays the name of the current directory path, a selection field, a scrollable directory listing and two buttons to terminate the routine. The user interacts with the dialog box in the standard manner, changing the directory being displayed, selecting an item from the directory list or typing in a user selection and then exiting via the OK OR CANCEL button.

The file selector library returns the filename selected or entered, in the buffer at addr_in(4), the directory path of the file in the buffer at addr_in(0) and whether the selection is ok or is to be cancelled. The application acts upon the information as required.

Entering the underscore into the directory string may cause the ST to crash

Window library

The window library routines permit the creation, opening, closing and deletion of windows to a maximum of eight active windows. The window parameters can be recovered or set, the window under the mouse cursor found, a flag set to indicate that a window is being updated and the size of a window determined.

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
WIND_CREATE	100	5	1	0	0	Allocate window size including border & ret'n window handle. W'dw open must set size < = to that allocated.
		int_in(0)	= window parts			
		(2)	= x coor		Of	
		(4)	= y coor		full	
		(6)	= width		size	
		(8)	= height		window	
		int_out(0)	= window handle (–ve, no windows available)			
WIND_OPEN	101	5	1	0	0	Open a window at its initial size and location, -not necessarily its full size.
		int_in(0)	= window handle			
		(2)	= x coor			
		(4)	= y coor		Window	
		(6)	= width		initial	
		(8)	= height		size	
		int_out(0)	= error 0 = yes, + ve no			
WIND_CLOSE	102	1	1	0	0	Close window, does not deallocate the window or handle.
		int_in(0)	= window handle			
		int_out(0)	= error 0 = yes, + ve no			
WIND_DELETE	103	1	1	0	0	Free space occupied by window and handle.
		int_in(0)	= window handle			
		int_out(0)	= error 0 = yes, + ve no			

Window parts (bit representation)

0x0001	Name	(name and title bar)
0x0002	Close	(close box)
0x0004	Full	(full box)
0x0008	Move	(move box)
0x0010	Info	(information line)
0x0020	Size	(size box)
0x0040	Uparrow	(up arrow)
0x0080	Dnarrow	(down arrow)
0x0100	Vslide	(vertical slider)
0x0200	Lfarrow	(left arrow)
0x0400	Rtarrow	(right arrow)
0x0800	Hslide	(horizontal slider)

Window library – continued

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
WIND_GET	104	2	5	0	0	Get window data specified field
		int_in(0)	= window handle			
		(2)	= get_field			
		int_out(0)	= error 0 = yes, + ve no			
		(2) =	Data			
		(4) =	specified			
		(6) =	by Get			
		(8) =	field			
WIND_SET	105	6	1	0	0	Set displayed window parameters
		int_in(0)	= window handle			
		(2)	= set_field			
		(4) =	Data			
		(6) =	specified			
		(8) =	by Set			
		($A) =	field			
		int_out(0)	= error 0 = yes, + ve no			

Get field	*int_out()* (2)	(4)	(6)	(8)	*Associated function*
4	x coor	y coor	width	height	window work area
5	x coor	y coor	width	height	current } size incl border
6	x coor	y coor	width	height	previous } title
7	x coor	y coor	width	height	max possible window size
8	1–1000	1 left, 1000 right			rel hor slider pos
9	1–1000	1 top, 1000 bottom			rel vert slider pos
10	handle				top window handle
11	x coor	y coor	width	height	1st rectangle in wind list
12	x coor	y coor	width	height	nxt rectangle in wind list
13	reservd				
15	1–1000	(–1 default min sq box)			rel horiz sld size
16	1–1000	(–1 default min sq box)			rel vert sld size
17					screen

Set field	*int_in()* (4)	(6)	(8)	($A)	*Associated function*
1	Parts				window components
2	Name pointer				address of name string
3	Info pointer				addr info line string
5	x coor	y coor	width	height	current window size
8	1–1000	1 left, 1000 right			rel hor slider pos
9	1–1000	1 top, 1000 bottom			rel vert slider pos
10	handle				top window handle
14	lo-word	hi-word	strtobj		GEM desktop to draw
15	1–1000	(–1 default min sq box)			rel hor slider size
16	1–1000	(–1 default min sq box)			rel vertical slider size
17					screen

Window library – continued

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
WIND_FIND	106	2	1	0	0	Find w'dw under mouse
	int_in(0)	= x coor] mouse				
	(2)	= y coor] position				
	int_out(0)	= window handle				
WIND_UPDATE	107	1	1	0	0	Flag about to u'd window
	int_in(0)	= update 0 = end, 1 = begin (window locking)				
		2 = end, 3 = begin (norm. mouse cntrl)				
	int_out(0)	= error 0_yes, + ve_no				
WIND_CALC	108	6	5	0	0	Ret w'dw border/work area
	intin(0)	= area 0 = work→border, 1 = border→work				
	(2)	= parts				
	(4)	= x coor				
	(6)	= y coor } border/work				To calculate work
	(8)	= width } area values				area, input border
	($A)	= height				area values.
	int_out(0)	= error 0_yes, + ve_no				To calc border
	(2)	= x coor				area, input work
	(4)	= y coor } work/border				area values.
	(6)	= width } area values				
	(8)	= height				

The desktop window is always present in the AES environment and supports a maximum of eight windows at a time. The AES screen manager handles all the user interaction outside the border area and the sizing, dragging and scrolling actions requested from within the border. The contents of the border area determine which of these functions are available.

Each user action sends a message through the message pipe to the applications EVNT_MESAG buffer where it is stacked on a first in-first out basis. In order to perform the requested function, the message must first be read and then the window management action may be either programmed to be performed or ignored. The assembler GEM program (Appendix L) demonstrates the effect of creating a window with the facilities, but not incorporating any code to handle the screen managers requests. The example also shows the parts handled by the screen manager, moving sliders, rubber boxing windows etc.

The application handles all activities within the windows work area.

Note that AES windows do not use the same coordinates as VDI areas:

AES x, y, width, height
VDI x1, y1, x2, y2

To create a window, the application calls WIND_CREATE defining the type (only those facilities that the application supports) and position of the window required, returning the window handle to be used in all subsequent actions on the window. An application call to WIND_CALC may be used to return the size of the window work area. A call to WIND_OPEN will get AES to draw the window's border area on the screen and send a message to the application to draw the windows work area.

WIND_SET calls are used to set the size and location of the vertical and horizontal sliders. If the window is resized, the application must decide if the preview rubber box size is valid. If not, the application may resize to the nearest valid size or display a warning dialog message. If valid, the application must issue a WIND_SET call to change the window size. A reduced window size does not require the work area to be redrawn, but if larger, GEM AES will send a message to the application to redraw the windows work area (EVNT_MESAG ID = 20).

The application is responsible for redrawing and updating the visible parts of its windows, which it divides into the smallest number of non-overlapping rectangles, found by a series of WIND_GET calls. Initially to the 'first' rectangle in the window list and subsequently to the 'next' rectangle until the returned width and height are both zero. Note that if the window is not covered, say by the control panel, that there will be only one rectangle.

Before updating the window, the application makes a WIND_UPDATE call to freeze all other rectangle lists and to prevent menus and alerts from being displayed during the update. On completion of the update, another WIND_UPDATE call permits further change to the display and rectangle lists.

To redraw the window work area, each rectangle in the rectangle list is compared with the update rectangle in turn, and any common portion redrawn.

To make a window active, the application (which must include an EVNT_MULTI call that includes a mouse button event) will receive a 'button pressed' message from the screen manager - the event occurred outside the active window and is therefore detected by the screen manager. The application calls WIND_FIND using the mouse x and y coordinates to obtain the handle of the window under the mouse. If it is the desktop, handle 0, a rubber box is drawn in expectation of the user selecting desktop icons. If the handle is that of an inactive window, the screen manager sends a message (EVNT_MESAG ID = 29) to request the window be brought to the top. The application calls WIND_SET to comply.

To close a window via the windows border or menu command, the screen manager sends a message to the application which should make a WIND_CLOSE call; a WIND_DELETE call will then free the handle.

Resource library

The resource library provides the interface between the application and its file resources, trees, objects, icons and pictures etc. providing the means to port an application to a different environment by simply changing the resource file data.

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
RSRC_LOAD	110	0	1	1	0	Allocate mem & load a resource file.
	int_out(0) = error 0 = yes, + ve no addr_in(0) = ASCII filename string address					
RSRC_FREE	111	0	1	0	0	Free the memory space allocated by rsrc_load.
	int_out(0) = error 0 = yes, + ve no					
RSRC_GADDR	112	2	1	0	1	Get address of data structure (object) loaded in memory.
	int_in(0) = type (2) = structure index int_out(0) = error 0 = yes, + ve no addr_out(0) = address of specified structure					

Function	*Op* $0	*Integers in* $2	*Integers out* $4	*Addresses in* $6	*Addresses out* $8	*Comments*
RSRC_SADDR	113	2	1	1	0	Store the address of a data structure.
	int_in(0) = type (2) = struct location index int_out(0) = error 0 = yes, + ve no addr_in(0) = address of the data structure					
RSRC_OBFIX	114	1	1	1	0	Convert objects location and size from character coordinates to pixels.
	int_in(0) = obj index int_out(0) = 1, reserved addr_in(0) = object tree address					

Type (of data structure)

0	tree	9	template string (tedinfo)
1	object	10	valid chars (tedinfo)
2	tedinfo	11	mask string (iconblk)
3	icon blk	12	data string (iconblk)
4	bitblk	13	text string (iconblk)
5	string	14	image pointer (bitblk)
6	imagedata	15	pointer address of free string
7	obspec	16	pointer address of free image
8	text string (tedinfo)		

To isolate an application from device, user and country specific data and provide program portability; GEM AES supports resource files that contain the variable parts of the application code.

To use a resource file, the application makes a RSRC_LOAD call to find the total file size in bytes, allocate the memory space for the resource file and update the file for screen resolution. The pointers to the object and the tree structures are also updated and the address of the tree array stored in the applications 'Global array'.

Access to the object library table pointers may be through RSRC_GADDR and RSRC_SADDR calls. The tree index may be accessed via FORM_DO and MENU_BAR calls among others.

RSRC_FREE deallocates the resource file memory and zeroes the tree array address in the Global array.

Resource files are generated using the Atari ST icon edit and resource utility program.

Shell library

The shell library routines enable one application to call another and keep track of command and command tails.

Function	*Op*	*Integers in*	*Integers out*	*Addresses in*	*Addresses out*	*Comments*
	$0	*$2*	*$4*	*$6*	*$8*	
SHEL_READ	120	0	1	2	0	Let application identify command that called it in fmt of GEM BDOS f.75
	int_out(0) = error 0 = yes, + ve no					
	addr_in(0) = buffer address of command string					
	(2) = buffer address of command tail					
SHEL_WRITE	121	3	1	2	0	Inform GEM which, if any, application to run, or exit GEM AES.
	int_in(0) = 0 = exit, 1 = run					
	(2) = graphic 0 = no, 1 = yes					
	(4) = GEM applic 0 = no, 1 = yes					
	int_out(0) = error 0 = yes, + ve no					
	addr_in(0) = new executable command file address					
	(2) = command tail address of next program					
SHEL_GET	122	1	1	1	0	Get data
	int_in(0) = length					
	int_out(0) = error code					
	addr_in(0) = buffer address					
SHEL_PUT	123	1	1	1	0	Put data
	int_in(0) = length					
	int_out(0) = error code					
	addr_in(0) = buffer address					
SHEL_FIND	124	0	1	1	0	Search for filename and return full DOS spec
	int_out(0) = error 0 = yes, + ve no					
	addr_in(0) = address 80 character buffer minimum					
	i/p search filename					
	o/p full DOS filename					
SHEL_ENVRN	125	0	1	2	0	Search for environment parameter and store address of following byte
	int_out(0) = 1, reserved					
	addr_in(0) = pointer to byte storage address					
	(2) = search parameter string					

The shell library routines use a single buffer containing the command and command tail that invoked the current application. A typical sequence to call and run another application might be:

Call SHEL_WRITE with a command, command tail and the home directory addresses; also define graphic/character or GEM/Not GEM application. On completion of the current application, the shell library will start the requested application.

Exit from GEM AES by making a SHEL_WRITE call with the int_in(0) parameter set to zero.

Utilities

INTELLIGENT KEYBOARD COMMAND INSTRUCTIONS

The Atari ST keyboard unit contains a 1MHz HD 6301 8-bit microprocessor with some on-board memory storage to maintain the time of day clock etc. The keyboard and its peripheral items, joystick and mouse may be initialized, monitored for position or status and the time of day clock read or set.

The intelligent keyboard (ikbd) communicates with the main processor over a 7.8 Kbit/s bidirectional serial link, sending individual keycodes or receiving instructions and returning status codes in packets of data through a pair of addresses, one for transmit and one for receive.

Characters can be read from the keyboard input queue in main system RAM, it is filled by an interrupt routine that transfers data from the ikbd to memory automatically. Characters are written to the keyboard by placing the character code in the keyboard data register after bit 1 of the keyboard command/status register is set.

Keycodes

The keyboard transmits make and break keycodes for each key press and release. Appendix D provides the codes for the individual keys, bit 7 being set for break and cleared for make.

Data packets

To differentiate the keyboard codes from the data packets transmitted to and from the ikbd to the main processor; the codes #$F6 to #$FF precede status information packets. The packets provide reports of mouse position and status, time of day and joystick status. The packets may be stored and used later, with the header byte removed, to restore the condition of the ikbd.

SEE 3-11 FOR TRAP#14 CALL TO WRITE STRING TO IKBD.

```
EXAMPLE :   PEA     SKBDDIS
            MOVE.W  #1, -(A7)       \ 2 PARAMETERS
            MOVE.W  # IKBDWS, -(A7)
            TRAP    #14
            ADDQ.L  #8, A7          \ TIDY UP
            RTS

IKBDWS      EQU     $19 \(OR #25)
SKBDDIS     DC.B    $12, $1A  \ SEE 3-95, 3-96.
```

IKBD commands

Input op code string	*Output databyte string*	*Function*	
#$80 #$01		Reset.	Return keyboard to power-up status without affecting clock. A break > 200ms also causes a reset.
#$07 00000aaa		Set mouse button action. default %00000000 1_press 1_release 0_button, 1_key type operation.	Mouse position report on (only relevent in absolute mode)
#$08		Set mouse relative position reporting (default).	Position packet generated asynchronously when threshold exceeded.
#$09 X msb X lsb Y msb Y lsb	Y maximum	Set mouse absolute positioning. X maximum mouse 'clicks' do not wrap,	Resets ikbd x and y coors. The x and y values in scaled ignore <0 & >max
#$0A X step Y step		Set mouse keycode mode.	Returns mouse motion in cursor keycodes instead of relative or absolute motion records.
#$0B X level Y level		Set mouse threshold.	Before move event is generated. default value 1. (Relative motion only)
#$0C X Y		Set mouse scale Set X and Y	Set X and Y scale factors for absolute mouse positioning – 'clicks' per coordinate change.

IKBD commands – continued

Input op code string	*Output databyte string*	*Function*	
#$0D	#$F7 0000xxxx	Interrogate mouse position	
		right button down since last interrogation right button up left button down left button up	since last report
	X msb X lsb Y msb Y lsb	X coor Y coor	Only valid in absolute mode, regardless of mouse button action setting.
#$0E #$00 X msb X lsb Y msb Y lsb		Load mouse position. filler X coor Y coor	Enables user to preset the internal absolute mouse position in scaled coordinate system
#$0F		Set Y = 0 at bottom	Set for relative and absolute mouse motions.
#$10		Set Y = 0 at top(default)	
#$11		Resume	Resume sending data back.
#$12		Disable mouse.	Stop mouse event reports. Resume on any mouse mode command.
#$13		Pause output	Stop sending further reports, queue them in a finite buffer.
#$14		Set joystick event reporting (default)	
#$15		Set joystick interrogation mode	Disable joystick event reporting, use interrogate to sense jstk state.
#$16		Joystick interrogation	Return a record of current joystick state.

IKBD commands – continued

Input op code string	*Output databyte string*	*Function*	
#$17 rate	000000ab aaaabbbb	Set joystick monitoring [packets of two] Joystick 1, Joystick 0 } Fire button Joystick 1, Joystick 0 } Pos'n Set ikbd to monitor serial command line and joystick,update time of day clock only	(sample rate of 0.01s)
#$18	ccccccc packed 8-bits/byte	Set fire button monitoring Set ikbd to monitor serial command line and fire button jstk 1, update time of day clock.	
#$19 Rx Ry Tx Ty Vx Vy		Set joystick keycode mode (provides a velocity autorepeat facility) Initial rate Tn Tn Tn; Final rate Vn Vn Rn length of time; times in 0.1s	(Joystick 0) If Rn zero only Vn matters.
#$1A		Disable joysticks. Disable any joystick event generation. Valid joystick commands resumes generation	
#$1B YY MM DD hh mm ss		Set time of day clock year (86, 87, 88 etc) month, day, hour, minute, second } Data sent in packed BCD format.	An invalid BCD digit does not alter the existing value.
#$1C	#$FC YY MM DD hh mm ss	Interrogate time of day clock year, month, day, hour, minute, second } Data in returned in packed BCD format.	

IKBD commands – continued

Input op code string	*Output databyte string*	*Function*	
#$20 Addr msb Addr lsb Numb (data)		Memory load ikbd controller address to be loaded. Number of data bytes (0-128)	
#$21 Addr msb Addr lsb	#$F6 #$20 data data data data data data	Memory read ikbd controller address to be read. 6 bytes of data starting at address (addr)	Status header memory access
#$22 Addr msb Addr lsb		Controller execute ikbd controller subroutine address	Allows main system to call an ikbd subroutine.
OR 80 with the set command	#$F6	Status inquiries. Get 8 byte data packet Strip packet header and return to recover status	

mode	#$7	#$8	#$9	#$A	#$0B	#$0C
param 1	code	0	Xmaxh	Xstep	Xthresh	Xtick
param 2	0	0	Xmaxl	Ystep	Ythresh	Ytick
param 3	0	0	Ymaxh	0	0	0
param 4	0	0	Ymaxl	0	0	0
param 5	0	0	0	0	0	0
param 6	0	0	0	0	0	0

ONLY 1 inquiry at a time.

#$F6 mode	Packet header						
	#$0F	#$10	#$12	#$14	#$15	#$19	#$1A
param 1						Rx	on
param 2	Inquiry			on		Ry	
param 3	returns					Tx	#$00
param 4	correct			#$00 off		Ty	off
param 5	mode					Vx	
param 6						Vy	

Not valid if in joystick or fire button monitoring mode.

When preceding a data packet returned from the keyboard, the special keycodes #$F6 to #$FF give the following meanings to the data packets:

Code Dec	Code Hex	Data packet function	
246	F6	Status report	
247	F7	Absolute mouse position record	
248	F8 to FB	Relative mouse position record 111110xx (xx = left-right button state) delta x, 2s complement delta y, 2s complement	
252	FC	Time of day (resolution of 1 second)	
253	FD	Joystick report header (both sticks)	
254	FE	x000yyyy } x = trigger	Joystick 0 event
255	FF	x000yyyy } y = stick position	Joystick 1 event

A-LINE ROUTINES

Atari ST programmers have access to the VDI primitives via the A-line exception routines; they provide a faster performance than the VDI routines, additional facilities and use less code to implement. The A-line routines may be mixed with VDI calls or used entirely on their own, but program portability to other systems will not be possible.

The A-line routines operate from a set of variables contained in a data table (Appendix F). The table is initialized by activating the A-line exception vector in passing the word #$A000. The programmer may then alter or insert variables into the data table and call the required function by passing the appropriate function call word.

```
dc.w     #$A000        ;initialize data table
move.w   #n,d(A0)      ;set function at offset d
                       ; to value n

dc.w     #$A00m        ;call function
```

Initialization creates the following pointers:

d0 = base address of A-line variables
a0 = base address of A-line variables
a1 = array of pointers to the 3 system font headers
a2 = array of pointers to the 15 A-line routines
(a2 is not returned correctly on disk based versions of TOS)

If VDI and AES are not used, the variables should be fairly static. If they are used, the variables may be changed, registers d0–d2 and a0–a2 will be trashed.

A-line routines

Function	Op $0	Pointpair in $2	Pointpair out $4	Integers in $6	Integers out $8	Comments
Put pixel	#$A001	1	0	1	0	Plot a pixel at x,y
	ptsin(0) = X_msbyte, Y_lsbyte					
	intin(0) = pixel value					
Get pixel	#$A002	1	0	0	0	Get the pixel at x,y
	ptsin(0) = X_msbyte, Y_lsbyte					return d0 = pixel value

A-Line routines – continued

Function	*Parameters*	*Comments*
Line #$A003		Draw a line between X1,Y1 and X2,Y2 The line is ALWAYS drawn from left to right and the mask applied left to right also- so watch the phase.
offset	$26 X1 coordinate $28 Y1 coordinate $2A X2 coordinate $2C Y2 coordinate $18 plane 0 } Bit value $1A plane 1 $1C plane 2 $1E plane 3 $22 line style mask ← $24 writing mode $20 –1 for XOR mode else ignore	Mask is word aligned pattern for horizontal lines. i.e. any bit of mask may used at the left-most endpoint.
output	$22 line style mask	Mask is rotated to align with rightmost endpoint.
Horiz Line #$A004		Draw a line between X1,Y1 and X2,Y1 The line is ALWAYS drawn from left to right
offset	26 X1 coordinate $28 Y1 coordinate $2A X2 coordinate $18 plane 0 } Bit value $1A plane 1 $1C plane 2 $1E plane 3 $24 writing mode $2E Fill pattern pointer $32 Fill pattern mask $34 Multi-plane fill flag	
output	none	
Filled rectangle #$A005		Draw a filled rectangle with upper lefthand corner X1,Y1 and lower righthand corner X2,Y2.
offset	26 X1 coordinate $28 Y1 coordinate $2A X2 coordinate $2C Y2 coordinate $18 plane 0 } Bit value $1A plane 1 $1C plane 2 $1E plane 3 $24 writing mode $2E Fill pattern pointer $32 Fill pattern mask	

Continued . . .

A-Line routines – continued

Function	*Parameters*	*Comments*
	$34 Multi-plane fill flag $36 Clipping flag $38 minimum X clipping value $3A maximum X clipping value $3C minimum Y clipping value $3E maximum Y clipping value	
output	none	
Line-by -line filled polygon #$A006	n - - - ptsin(0) = X,Y array of polygon vertices	Draw 1 scan-line of a filled polygon.
offset	$28 Y1 coordinate $18 plane 0 } Bit value $1A plane 1 $1C plane 2 $1E plane 3 $24 writing mode $2E Fill pattern pntr $32 Fill pattern mask $34 Multi-plane fill flag $36 Clipping flag $38 minimum X clipping value $3A maximum X clipping value $3C minimum Y clipping value $3E maximum Y clipping value	Polygon X1,Y1...Xn,Yn...X1,Y1 Start point must be repeated at the end of the list Y1 is the Y coor of line to fill.
output	none	X1 and X2 trashed on return
Bitblt #$A007		Bit block transfer
input	a6 = i/p parameter table pointer	
output	none	
Textblt #$A008		Perform a Text block transfer of 1 character.
offset	24 = writing mode $6A = Foreground } Text colour $72 = Background $54 = Pointer } Font form $58 = Width $48 = X coor $4A = Y coor $4C = X coor } Character on screen $4E = Y coor $50 = width } Character $52 = height	Writing mode 0-3 VDI modes 4-19 Bitblt modes

Continued . . .

A-Line routines – continued

Function	*Parameters*	*Comments*
	$5A = Style flag $5C = Lighten text mask $5E = Skew text mask $60 = Thickening txt width $62 = above] Char offset $64 = below] from baseline $66 = Scaling flag (0 = none) $40 = Accumulator for x dda $42 = Textblt scale factor $44 = Scale dir (down > 0) $68 = Char rotation vector $46 = Font status $6C = Special effects buffer pointer $70 = Scaling buffer offset in above pointer output none	
Show mouse	#$A009 input none output none	Show the mouse, if # of 'show' calls >= # of 'hide' calls.
Hide mouse	#$A00A input none output none	Hide the mouse, if # of 'hide' calls exceeds # of 'show' calls.
Transform mouse	#$A00B Cntrl $E = Addr.L src MFDB Cntrl $12 = Addr.L dest MFDB output none	Transform mouse form
Undraw sprite	#$A00C input a2 = sprite slave blk pntr The sprite save block saves the screen underneath the sprite and is (10bytes + 64 × # planes) bytes in size. output none	Undraw previously drawn sprite *** a6 smashed ***

A-Line routines – continued

Function	*Parameters*	*Comments*
Draw sprite	#$A00D input d0 = X hot spot d1 = Y hot spot a0 = pointer sprite definition block a2 = pointer sprite save block output none	Draw a sprite (Funct not avail'bl directly on disk-based vs of TOS) *** a6 smashed ***
Copy raster form	#$A00E cntrl $E = Addr.L (source MFDB) cntrl $12 = Addr.L (destination MFDB) output none	Copy a raster from source to destination.

A-line variables table

offset		*Function*	
$00	0	Number of video planes	Can produce special effects.
$02	2	Number of bytes/video line	
$04	4	Pointer to `cntrl` array	
$08	8	Pointer to `intin` array	
$0C	12	Pointer to `ptsin` array	
$10	16	Pointer to `intout` array	
$14	20	Pointer to `ptsout` array	
$18	24	Bit plane_0	current colour value
$1A	26	Bit plane_1	
$1C	28	Bit plane_2	
$1E	30	Bit plane_3	
$20	32	–1	
$22	34	VDI line style equivalent	
$24	36	Writing mode: 0 = replace, 1 = transparent 2 = XOR mode, 3 = inverse transparent	
$26	38	X1 coordinate	
$28	40	Y1 coordinate	
$2A	42	X2 coordinate	
$2C	44	Y2 coordinate	
$2E	46	Pointer to current fill pattern	
$32	50	Fill pattern mask (length of pattern)	

Continued . . .

A-Line routines – continued

Function		*Parameters*	*Comments*
$34	52	Multi-plane fill pattern 0_current fill pattern is single plane 1_current fill pattern is multi-plane	
$36	54	Clipping flag 0 = no clipping	
$38	56	Minimum x clipping value	
$3A	58	Minimum y clipping value	
$3C	60	Maximum x clipping value	
$3E	62	Maximum y clipping value	
$40	64	Accumulator for textblt x dda initialize to 8000H before each call	
$42	66	Textblt scale factor	
$44	68	Scale direction 0 = down	
$46	70	Font status 1 = current font monospaced & may be thickened 0 = may not be thickened to increase font width	
$48	72	X coor of character in font form	
$4A	74	Y coor of character in font form (typically 0)	
$4C	76	X coor of character on screen	
$4E	78	Y coor of character on screen	
$50	80	Character width	
$52	82	Character height	
$54	84	Pointer to start of font data (font form)	
$58	88	Width of font form	
$5A	90	Style bit 0 = Thicken, bit 1 = lighten, bit 2 = skew bit 3 = underline (ignored), bit 4 = outline	
$5C	92	Lighten text mask	
$5E	94	Skew text mask	
$60	96	Text thickening additional width	
$62	98	Offset above character baseline for skew	
$64	100	Offset below character baseline for skew	
$66	102	Scaling flag 0 = no scaling	
$68	104	Character rotation vector 0 = horizontal 900 = vertical down etc.	
$6A	106	Text foreground colour	
$6C	108	Special effects buffer pointer	
$70	112	Scaling buffer offset in above buffer	
$72	114	Text background colour	(RAM VDI only)
$74	116	Copy raster form type flag 0 = opaque type n-plane source to n-plane destination bitblt write mode <>0 = transparent type 1-plane source to n-plane destination VDI write mode	(RAM VDI only)
$76	118	Abort fill routine pointer (Function not available on disk based versions of TOS)	

Sprite definition block

offset

$00	X offset of hot-spot	
$02	Y offset of hot-spot	
$04	Format flag	
$06	Background	Colour table index
$08	Foreground	Colour table index
$0A	Interleaved	Background line 0
$0C	Background / Foreground	Foreground line 0
—		
$4A	image (32 words)	Foreground line 16
$4C		

Format flag

::

+ve		*–ve*		*colour plotted*
Fg	*Bg*	*Fg*	*Bg*	
0	0	0	0	Transparent
0	1	0	1	Background
1	1	1	1	Foreground
1	0			Foreground
		1	0	XOR screen

Memory form definition block (MFDB)

offset

$00	Memory pointer	32-bit address of pixel 0,0
$04	Width	Raster area dimensions
$08	Height	Raster area dimensions
$0C	Word width	Pixel width/word size
$10	Format flag	1 = standard, 0 = device specific
$14	Memory planes	No. planes in raster area
$18		Three reserved words
$1C		
$20		
$24		

BITBLT table used in block transfers

Parameter block length must be 76 bytes, the first 52 bytes being set by the user and the remainder by the blt. Address register A6 is used as a pointer to the table, a point that 'C' programmers should note.

0	b_width		width } of block in
2	b_height		height } pixels
* 4	#planes		# of cosecutive planes to blt
* 6	fg_col		foreground colour high bit } logic op
* 8	bg_col		background colour low bit } index
10	op_table	$A	logic op – Table of 4 raster op code bytes,
11		$B	each containing one of sixteen logical
12		$C	operations. They are indexed by fg × 2 + bg for
13		$D	each plane.(see below)
14	s_xmin	$E	minimum source x
16	s_ymin	$10	minimum source y
18	s_form	$12	source form base address (word b'dry)
22	s_nxwd	$16	word in line } next offset (2 = hi,4 = med,*8 = lo)*
24	s_nxln	$18	line in plane } in bytes (90 = hi,160 = med/lo)
26	s_nxpl	$1A	next plane offset from current (always 2)
28	d_xmin	$1C	minimum destination x
30	d_ymin	$1D	minimum destination y
32	d_form	$20	destination form base addr (word boundary)
36	d_nxwd	$24	word in line } next offset (2 = hi,4 = med,8 = lo)
38	d_nxln	$26	line in plane } in bytes (90 = hi,160 = med/lo)
40	d_nxpl	$28	next plane offset from current (always 2)
* 42	p_addr	$2A	addr of pattern buffer (0 = none)
			A word size repetitive, word aligned pattern that is ANDed with the source before being logically combined with the destination.
46	p_nxln	$2D	next line in pattern } offset (2, 4, 6 etc)
48	p_nxpl	$30	next plane in pattern } in bytes (0 = 1 plane)
50	p_mask	$32	pattern index mask length

* may be altered during bitblt execution

The source bit defined by s_xmin, s_ymin, b_width, b_height is transfered to destination d_xmin, d_ymin by the number of planes iterations (#planes). There is no clipping or check that bit blocks are within the encompassing memory forms.

Logic Table

			Fg	*bg*
10	Op_0	$A	0	0
11	Op_1	$B	0	1
12	Op_2	$C	1	0
13	Op_3	$D	1	1

The logic operation bytes (see Chapter 3) specify the effect of foreground and background colour bits on the current plane.

Interrupt handler

The standard system interrupt is level 2, vector $68 (104) and takes the following sequence every interrupt:

Vertical blank interrupt (VBI)

Order	*Function*	*Sys variable*	
1	Increment the frame counter	FRCLOCK.L	$466
2	Test for mutual exclusion if = < 0 return	VBSLEM.W	$452
3	Save all the registers on stack		
4	Increment 'Vblank counter'	VBCLOCK.L	$462
5	Test for high resolution mode if shftmd < 2 then goto 6 test for low resolution monitor attatched, if yes set mode to zero or	SHFTMD.W DEFSHFTMD.B	$44C $44A
6	Call cursor blink routine		
7	Test for new colour pallette if colourptr = 0 then goto 8 Load pallette with 16 words pointed to by colourptr and then zero it.	COLORPTR.L	$45A
8	Test for new screen if screenptr = 0 then goto 9 Set screen physical base to screen pointer and then zero pointer.	SCREENPTR.L	$45E
9	Run deferred VBI vectors # of deferred VBI vectors Pointer to VBI vector array	nvbls.W vblqueue.L	$454 $456
10	Return		

There are eight VBI vectors available in the default array, the first is reserved for GEM's VBI code. Pointers to new handlers are placed in the spare slots. Handler code ends in RTS and may use any register except the user stack pointer. Larger arrays can be allocated by redefining nvbls and vblqueue, copying the current vectors to the new array. An application that returns should tidy up the VBI queue.

Appendix A
System variables

The following tables present the system variables in low supervisor space $0 to $7FF (0 to 2047):

Exception vectors

10	$000	Reset initial SSP value	
4	$004	Reset initial PC address	
8	$008	Bus error	Dump state
12	$00C	Address error	and terminate
16	$010	Illegal instr.	routine pointer
20	$014	Divide by zero Pointer to an RTE	
24	$018	Chk instr.	Dump state
28	$01C	Trapv instr.	and terminate
32	$020	Privilege viol	routine pointer
36	$024	Trace mode	
40	$028	Line 1010	A-line routine pointer
44	$02C	Line 1111	Used by AES
48	$030	Unassigned	
52	$034	Coprocessor protocol violation (MC68020)	
56	$038	Format error (MC68020)	
60	$03C	Uninitialized interrupt vector	
64	$040	Unassigned	
*	–	–	Reserved
82	$05C	Unassigned	
96	$060	Spurious interrupt (Hacked to level 3)	
100	$064	Int level 1	(Used if user wants Hblanks)
104	$068	Int level 2	Horizontal blank sync (Hblank)
108	$06C	Int level 3	Normal processor interrupt level
112	$070	Int level 4	Vertical blank sync (Vblank)
116	$074	Int level 5	
120	$078	Int level 6	MK68901 MFP interrupts
124	$07C	Int level 7	Non maskable interrupt
128	$080	Trap #0	
132	$084	Trap #1	GEM DOS interface calls
136	$088	Trap #2	Extended DOS calls
140	$08C	Trap #3	
–	–	–	
176	$0B0	Trap #12	
180	$0B4	Trap #13	GEM BIOS calls
184	$0B8	Trap #14	Atari extended BIOS calls
188	$0BC	Trap #15	
192	$0C0	Unassigned	
*	–	–	Reserved
252	$0FC	Unassigned	

MFP hardware bound interrupt vectors

256*	$100	Parallel port interrupt_0 (Centronics busy)
260*	$104	RS232 carrier detect (dcd) interrupt_1
264*	$108	RS232 clear to send (cts) interrupt_2
268*	$10C	Graphics blt done interrupt_3
272*	$110	RS232 baud rate generator (Timer D)
276	$114	200Hz system clock (Timer C)
280	$118	Keyboard/MIDI (6850) interrupt_4
284*	$11C	Polled fdc/_hdc interrupt_5
288*	$120	Horizontal blank counter (Timer B)
292	$124	RS232 transmit error interrupt
296	$128	RS232 transmit buffer empty interrupt
300	$12C	RS232 receive error interrupt
304	$130	RS232 receive buffer full interrupt
308*	$134	User/application (Timer A)
312*	$138	RS232 ring indicator interrupt_6
316*	$13C	Polled monochrome monitor detect interrupt_7
320	$140	
*	–	–
508	$1FF	

* Initially disabled — Priority levels (7 high)

The polled fdc/_hdc interrupt *must* be disabled on return.

Application interrupts

512	$200		
*	–	–	Reserved for OEMs
892	$37C		

After an uncaught trap, the processor state is dumped as follows:

Processor state

896	$380	proc_lives	Processor state saved if system variable set to $12345678
900	$384	proc_regs	D0–D7/A0–A6, A7_ssp
964	$3C4	proc_pc	First byte exception number
968	$3C8	proc_usp	USP
972	$3CC	proc_stk	sixteen words of superstack

The above values are not overwritten by a system reset, but are by a further crash.

System variables

1024	$400	L	etv_timer	Timer handoff (logical vector $100)
1028	$404	L	etv_critic	Critical error handoff vector ($101)
1032	$408	L	etv_term	Process terminate handoff vect ($102)
1036	$40C	5 × L	etv_xtra	Space for res'd logical vectors ($103–$107)
1056	$420	L	memvalid	#$752019F3 (cold start ok)
1060	$424	B	memcntlr	memory controller low nibble 0 = 128K, 4 = 512K, (0 = 256K, 5 = 1MB 2 banks)
1062	$426	L	resvalid	#$31415926 to jump through resvector
1066	$42A	L	resvector	System reset bailout vector
1070	$42E	L	phystop	Phys RAMtop (points 1st unusable byte)
1074	$432	L	_membot	Available memory bottom (getmpb uses)
1078	$436	L	_memtop	Available memory top (getmpb uses)
1082	$43A	L	memval2	#$237698AA
1086	$43E	W	flock	Floppy FIFO lock variable
1088	$440	W	seekrate	0 = 6ms, 1 = 12ms, 2 = 2ms, 3 = 3ms default
1090	$442	W	_timr_ms	20 (#$14) system timer calibration
1092	$444	W	_fverify	0 = no write–verify else verify (default)
1094	$446	W	_bootdev	System boot device number
1096	$448	W	palmode	0 = NTSC, 60Hz else PAL, 50Hz
1098	$44A	B	defshftmd	Default video res if monitor changed
1100	$44C	B	sshiftmd	Shadow shiftmd hardware register 0 = 320 × 200 × 4 1 = 640 × 200 × 2 2 = 640 × 400 × 1
1102	$44E	L	_v_bas_ad	Screen mem base pntr (32K contiguous) on a 512 byte boundary
1106	$452	W	vblsem	Vertical blank mutual exclusion semaphore 1_vblank enabled
1108	$454	W	nvbls	8 (No. longwords vblqueue points to)
1110	$456	L	_vblqueue	Vblank handler pointer to pointers
1114	$45A	L	colorptr	0 null else pointer to 16 word vector for hardware pallette next vblank
1118	$45E	L	screenpt	Pointer to screen base next vblank or 0
1122	$462	L	_vbclock	Vertical blank interrupt count
1126	$466	L	_frclock	Count vblank interrupts not vblsem'd

System variables – continued

1130	$46A	L	hdv_init	Hard disk initialize vector else zero
1134	$46E	L	swv_vec	'Monitor changed' vector to follow
1138	$472	L	hdv_bpb	Hard disk vector to return bpb else 0
1142	$476	L	hdv_rw	Hard disk rd/wr routine vector else 0
1146	$47A	L	hdv_boot	Hard disk boot routine vector else 0
1150	$47E	L	hdv_mediach	Disk media change rout vector else 0
1154	$482	W	_cmdload	<>0 load & exe COMMAND.PRG (boot dev)
1156	$484	B	conterm	Attribute bits for console sys, bit: 0_bell on (^G), 1_keyrepeat 2_keyclick, 3_bios conin() function, kbshft in bits 24–31 of D0.L
1157	$485	B		reserved
1158	$486	L	trp14ret	Saved trap 14 return address
1162	$48A	L	criticret	Saved return address for etv_critic
1166	$48E	L	themd	GEMDOS memory descriptors (don't change) Structure MD m_link — Next MD/null m_start — Start of TPA m_length — Byte size of TPA m_own — MD's owner/null
1182	$49E	W	_md	?
1186	$4A2	L	savptr	BIOS register save area pointer
1190	$4A6	W	_nflops	No floppies attached 0, 1 or 2
1192	$4A8	L	con_state	State of conout() parser
1196	$4AC	W	save_row	Save row# for x–y addressing
1198	$4AE	L	sav_contxt	Pointer to saved processor context
1202	$4B2	L	_bufl	GEMDOS two buffer-list pointers 1st buffers data sectors 2nd buffers FAT and DIR sectors Structure BCB b_link — Next BCB b_bufdrv — Drive#/–1 b_buftyp — Buffer type b_bufrec — Record # cached b_dirty — Dirty flag b_dm — Drive media descriptor b_bufr — Buffer pointer

System variables – continued

1210	$4BA	L	_hz_200	Raw 200Hz timer tick
1214	$4BE	L	the_env	Default environment string $00000000
1218	$4C2	L	_drvbits	32 bit vector of live block devices
1222	$4C6	L	_dskbufp	Pointer to common disk buffer, 1 Kbyte in systems BSS. (Do not use by an interrupt routine)
1226	$4CA	L	_autopath	Pointer to autoexec path (or null)
1230	$4CE	8 × L	_vbl_list	Initial vblqueue
*	–	–		
*	–	–		
1262	$4EE	W	_prt_cnt	Initially –1., Alt_Help increments
1264	$4F0	W	_prtabt	Printer abort flag
1266	$4F2	L	_sysbase	Base of OS pointer (RAM or ROM)
1270	$4F6	L	_shell_p	Global shell info pointer
1274	$4FA	L	end_os	Pointer to end of OS memory usage *
1278	$4FE	L	exec_os	Pointer to shell addr to exec on startup (normally 1st byte of AES text seg).
2048	$800			Start of user RAM

* THIS COINCIDES WITH _MEMBOT (AT 1076)
AND M_START (AT 1166)
(IS THIS COINCIDENCE)

Appendix B
Configuration registers

Configuration Registers (one/_zero)

MEMORY

16744452	FF8004	r/w	|....xxxx|	Memory configurations
			Bank 0	Bank 1 (not used)
			0 128Kbyte	128Kbyte
			1 128Kbyte	512Kbyte
			2 128Kbyte	2Mbyte
			3 reserved	
			4 512Kbyte	128Kbyte
			5 512Kbyte	512Kbyte
			6 512Kbyte	2Mbyte
			7 reserved	
			8 2Mbyte	128Kbyte
			9 2Mbyte	512Kbyte
			10 2Mbyte	2Mbyte
			11 reserved	
			12+ reserved	

DISPLAY

16745061	FF8201	r/w	|xxxxxxxx|	Video base high
16745063	FF8203	r/w	|xxxxxxxx|	Video base low
16745065	FF8205	r	|xxxxxxxx|	Video address counter high
16745067	FF8207	r	|xxxxxxxx|	Video address counter mid
16745069	FF8209	r	|xxxxxxxx|	Video address counter low
16745071	FF820A	r/w	|.....xx|	***Sync mode***
				External/_Internal sync
				50Hz/_60Hz field rate
16745124	FF8240	r/w	|....xxx.xxx.xxx|	Pallette color 0/0 (border)
				Invert /_normal b&w
				Blue
				Green
				Red
16745126	FF8242	r/w	|.....xxx.xxx.xxx|	Pallette color 1/1
16745128	FF8244	r/w	|.....xxx.xxx.xxx|	Pallette color 2/2
16745130	FF8246	r/w	|.....xxx.xxx.xxx|	Pallette color 3/3
16745132	FF8248	r/w	|.....xxx.xxx.xxx|	Pallette color 4
16745134	FF824A	r/w	|.....xxx.xxx.xxx|	Pallette color 5
16745136	FF824C	r/w	|.....xxx.xxx.xxx|	Pallette color 6
16745138	FF824E	r/w	|.....xxx.xxx.xxx|	Pallette color 7
16745140	FF8250	r/w	|.....xxx.xxx.xxx|	Pallette color 8
16745142	FF8252	r/w	|.....xxx.xxx.xxx|	Pallette color 9
16745144	FF8254	r/w	|.....xxx.xxx.xxx|	Pallette color 10
16745146	FF8256	r/w	|.....xxx.xxx.xxx|	Pallette color 11

Continued . . .

Configuration registers – continued

16745148	FF8258	r/w	|.....xxx.xxx.xxx|	Pallette color 12
16745150	FF825A	r/w	|.....xxx.xxx.xxx|	Pallette color 13
16745152	FF825C	r/w	|.....xxx.xxx.xxx|	Pallette color 14
16745154	FF825E	r/w	|.....xxx.xxx.xxx|	Pallette color 15
16745156	FF8260	r/w	|.....xx|	***Shift mode***
				0 320 × 200, 4 plane
				1 640 × 200, 2 plane
				2 640 × 400, 1 plane
				3 Reserved

RESERVED

16745572	FF8400		|.....|	reserved

DMA/DISK

16746084	FF8600		|.....|	reserved
16746086	FF8602		|.....|	reserved
16746088	FF8604	r/w	|.....xxxxxxxx|	Disk controller data access
16746090	FF8606	r	|.....xxx|	DMA status (mode control)
				_Error
				_Sector count zero
				_Data request inactive
	FF8606	w	|.....xxxxxxxx.|	DMA mode control
				A0 – WD1772 register
				A1 – WD1772 register
				HDC/_FDC reg select
				Sect count reg selct
				0 reserved
				Disable/_enable DMA
				FDC/_HDC
				Write/_Read
16746093	FF8609	r/w	|xxxxxxxx|	DMA base & counter high
16746095	FF860B	r/w	|xxxxxxxx|	DMA base & counter mid
16746097	FF860D	r/w	|xxxxxxxx|	DMA base & counter low

Configuration registers – continued

SOUND

16746596	FF8800	r	|xxxxxxxx|		***PSG read data***
					I/O port B, Par i/f data
		w	|xxxxxxxx|		***PSG register select***
					register number
			8 bit	0	Channel A fine tune
			4 bit	1	Channel A coarse tune
			8 bit	2	Channel B fine tune
			4 bit	3	Channel B coarse tune
			8 bit	4	Channel C fine tune
			4 bit	5	Channel C coarse tune
			5 bit	6	Noise generator control
			8 bit	7	Mixer control–I/O enable
			5 bit	8	Channel A amplitude
			5 bit	9	Channel B amplitude
			5 bit	10	Channel C amplitude
			8 bit	11	Envelope period fine tune
			8 bit	12	Envel period coarse tune
			4 bit	13	Envelope shape
				14	I/O port A (output only)
				15	I/O port B (Centronics o/p)
16746598	FF8802	w	|xxxxxxxx|		***PSG write data,***
					I/O port A
					Floppy side 0/_side 1 sel
					Floppy_drive 0 select
					Floppy_drive 1 select
					RS232 RTS
					RS232 DTR
					Centronics STROBE
					General purpose output
					Reserved
		r/w	|xxxxxxxx|		I/O port B, Par i/f data

Configuration registers – continued

MK68901

16775681	FFFA01	|xxxxxxxx|	***MFP G.P. I/O***
			Parallel port status
			WD1772 active
			Interrupt
			Mono monitor
16775683	FFFA03	|xxxxxxxx|	MFP active edge
16775685	FFFA05	|xxxxxxxx|	MFP data direction
16775687	FFFA07	|xxxxxxxx|	MFP interrupt enable A
16775689	FFFA09	|xxxxxxxx|	MFP interrupt enable B
16775691	FFFA0B	|xxxxxxxx|	MFP interrupt pending A
16775693	FFFA0D	|xxxxxxxx|	MFP interrupt pending B
16775695	FFFA0F	|xxxxxxxx|	MFP intrpt in-service A
16775697	FFFA11	|xxxxxxxx|	MFP intrpt in-service B
16775699	FFFA13	|xxxxxxxx|	MFP interrupt mask A
16775701	FFFA15	|xxxxxxxx|	MFP interrupt mask B
16775703	FFFA17	|xxxxxxxx|	MFP vector base
16775705	FFFA19	|xxxxxxxx|	MFP timer A control
16775707	FFFA1B	|xxxxxxxx|	MFP timer B control
16775709	FFFA1D	|xxxxxxxx|	MFP timers C & D control
16775711	FFFA1F	|xxxxxxxx|	MFP timer A data
16775713	FFFA21	|xxxxxxxx|	MFP timer B data
16775715	FFFA23	|xxxxxxxx|	MFP timer C data
16775717	FFFA25	|xxxxxxxx|	MFP timer D data
16775719	FFFA27	|xxxxxxxx|	MFP sync character
16775721	FFFA29	|xxxxxxxx|	MFP USART control reg
16775723	FFFA2B	|xxxxxxxx|	MFP receiver status
16775725	FFFA2D	|xxxxxxxx|	MFP transmitter status
16775727	FFFA2F	|xxxxxxxx|	MFP USART data

MC6850

16776192	FFFC00	|xxxxxxxx|	Keyboard ACIA control
16776194	FFFC02	|xxxxxxxx|	Keyboard data
16776196	FFFC04	|xxxxxxxx|	Midi ACIA control
16776198	FFFC06	|xxxxxxxx|	Midi data

Appendix C
Printer and terminal escape codes

In general an Atari printer that is designed to work with the ST will provide the most suitable path to trouble free computer/printer interfacing and the production of hard copy printout and screen dumps. Where a printer of another manufacture is to be used, the following information may be of use:

If screen dumps are required, the code 1B 4C (27 76 decimal) should be recognized as 'double density bit image mode' for printing 960 dots per line at 120 dots per inch on 8" wide paper (the dump is virtually the same size as the monitor screen display) or code 1B 59 (27 89 decimal) for the wider paper screen dumps.

It may reasonably be assumed that whatever word processor you employ, it will provide the necessary print configuration file to make available the printers facilities. Double clicking a non-executable file icon to print its contents should not cause problems as control codes are not sent within the text. The ST does however precede the file with the code to select draft or NLQ (near letter quality) print, i.e. ESC,"x",n.

Some serial printers are restricted to 2400 and 600 baud operation, the ST supports neither rate without recourse to 'C' or assembly language programming.

TYPICAL EPSON PRINTER CODES

Code Dec	Hex	Ascii Mnem	Function
0	00	**NUL**	
1	01	**SOH**	
2	02	**STX**	
3	03	**ETX**	
4	04	**EOT**	
5	05	**ENQ**	
6	06	**ACK**	
7	07	**BEL**	Bell
8	08	**BS**	Backspace
9	09	**HT**	Tab horizontal
10	0A	**LF**	Line feed
11	0B	**VT**	Tab vertical
12	0C	**FF**	Form feed
13	0D	**CR**	Carr. Return
14	0E	**SO**	* Enlarged
15	0F	**SI**	Condensed
16	10	**DLE**	
17	11	**DC1**	on-line printer
18	12	**DC2**	Condensed off
19	13	**DC3**	Off-line printer
20	14	**DC4**	* Enlarged off
21	15	**NAK**	
22	16	**SYN**	
23	17	**ETB**	
24	18	**CAN**	Clear print buffer
25	19	**EM**	
26	1A	**SUB**	
27	1B	**ESC**	
28	1C	**FS**	
29	1D	**GS**	
30	1E	**RS**	
31	1F	**US**	
32	20		Printable ASCII codes
!			
!			
127	7F		

Code Dec	Hex	ASCII Char	ESCape code functions
Dec	Hex	Char	ESCape code
32	20	|	
33	21	!	Combine print modes
34	22	"	
35	23	#	
36	24	$	
37	25	%	Sel ROM/user char set
38	26	&	Define user characters
39	27	'	
40	28	(	
41	29	)	
42	2A	*	Select graphics mode
43	2B	+	
44	2C	,	
45	2D	-	Underline on/off
46	2E	.	
47	2F	/	Sel vert tab chan
48	30	0	Set 1/8 inch LF
49	31	1	Set 7/72 inch LF
50	32	2	Set 1/6 inch LF
51	33	3	Set n/216 inch LF
52	34	4	Italic on
53	35	5	Italic off
54	36	6	
55	37	7	
56	38	8	Detect paper-out on
57	39	9	Detect paper-out off
58	3A	:	Copy ROM char to RAM
59	3B	;	
60	3C	<	* Unidirection print
61	3D	=	
62	3E	>	
63	3F	?	Redef. graphic mode
64	40	@	Initialize printer
65	41	A	Set n/72 inch LF
66	42	B	Set vertical Tabs
67	43	C	n Set form length
68	44	D	Set horizontal Tabs
69	45	E	Bold on

* for one line only

Typical Epson printer codes – continued

Code Dec	Hex	ASCII Char	ESC code functions	Code Dec	Hex	ASCII Char	ESCape code functions
70	46	**F**	Bold off	85	55	**U**	Unidirection on/off
71	47	**G**	Double strike on	87	57	**W**	Enlarged on/off
				89	59	**Y**	120 dpi bitimage-fast
72	48	**H**	Double strike off	90	60	**Z**	240 dpi bitimage
73	49	**I**		94	64	^	Set 9 pin bit image
74	4A	**J**	LF n/216 inch	97	61	**a**	Set NLQ justify
75	4B	**K**	60 dpi bitimage	98	62	**b**	Set vert tabs channels
76	4C	**L**	120 dpi bitimage	101	65	**e**	Set hor/ver Tab increment
77	4D	**M**	Elite on	102	66	**f**	Paperfeed/Tab execute
78	4E	**N**	Skip perforation on	108	6B	**l**	Set left margin
79	4F	**O**	Skip perforation off	109	6C	**m**	Special character generator
80	50	**P**	Pica on/Elite off	112	70	**p**	Proportional on/off
81	51	**Q**	Set right column	115	73	**s**	Half speed on/off
82	52	**R**	Select character set	120	78	**x**	Select draft/NLQ mode
83	53	**S**	Super/subscript on	127	7F	**del**	Cancel last char
84	54	**T**	Super/subscript off				

VT52 TERMINAL ESCAPE CODES

The following BIOS bconout() functions simulate a VT52 terminal, with extensions for color, screen wrap etc.

Escape	*Function*	*Comments*
A	Cursor up	Up one line, no effect if at top
B	Cursor down	Down one line, no effect if at bottom
C	Cursor right	Right one position, no effect if at edge
D	Cursor left	Left one position, no effect if at edge
E	Clear screen	Clear screen and home cursor to column 0, row 0
H	Home cursor	Home cursor to column 0,row 0
I	Cursor up	Up one line, if at top scroll
J	Erase to eop	Erase to end of page from and including cursor position
K	Clear to eol	Clear to end of line from cursor of line position
L	Insert line	Insert blank line with cursor at start of line. Move current line down
M	Delete line	Delete cursor line and move remaining lines up one, put blank at bottom.
Y,r,c	Cursor r,c	Position cursor at row r column c
b,f	fg'd colour f	Color is the 4 lsb of color byte
c,b	bg'd color b	Color is the 4 lsb of color byte
d	Erase to start of page	Erase to start of page including the current cursor position
e	Show cursor	Show cursor
f	Hide cursor	Hide cursor
j	Save cursor	Save the cursor position
k	Restore cursor	Restore cursor, home if no saved position
l	Erase line	Erase line and move cursor to left edge
o	Erase to line start	Erase to start of line from and including the cursor
p	Reverse video	Enter reverse video mode
q	Normal video	Exit reverse video mode
v	Wrap at end of line	Wrap at end of line and scroll up if necessary
w	Discard end of line	Overprint line end character with the next character

Appendix D
Keycode definitions

ASCII CODES 0–127

Dec	*Ascii*	*Dec*	*Ascii*	*Dec*	*Ascii*	*Dec*	*Ascii*
0	NUL	32	SPACE	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

GSX COMPATIBLE KEYSCAN CODES

Code Dec	*Code Hex*	*Keytop*	*Code Dec*	*Code Hex*	*Keytop*	*Code Dec*	*Code Hex*	*Keytop*
1	01	ESC	41	29	'	81	51	*n.u*
2	02	1	42	2A	L SHIFT	82	52	INSERT
3	03	2	43	2B	\	83	53	DEL
4	04	3	44	2C	Z	84	54	*n.u*
5	05	4	45	2D	X			
6	06	5	46	2E	C			
7	07	6	47	2F	V	95	5F	*n.u*
8	08	7	48	30	B	96	60	ISO key
9	09	8	49	31	N	97	61	UNDO
10	0A	9	50	32	M	98	62	HELP
11	0B	0	51	33	,	99	63	*kpd* (
12	0C	-	52	34	.	100	64	*kpd*)
13	0D		53	35	/	101	65	*kpd* /
14	0E	BS	54	36	R SHIFT	102	66	*kpd* *
15	0F	TAB	55	37	*n.u*	103	67	*kpd* 7
16	10	Q	56	38	ALT	104	68	*kpd* 8
17	11	W	57	39	SPACE	105	69	*kpd* 9
18	12	E	58	3A	CAPS LOCK	106	6A	*kpd* 4
19	13	R	59	3B	F1	107	6B	*kpd* 5
20	14	T	60	3C	F2	108	6C	*kpd* 6
21	15	Y	61	3D	F3	109	6D	*kpd* 1
22	16	U	62	3E	F4	110	6E	*kpd* 2
23	17	I	63	3F	F5	111	6F	*kpd* 3
24	18	O	64	40	F6	112	70	*kpd* 0
25	19	P	65	41	F7	113	71	*kpd* .
26	1A	[	66	42	F8	114	72	*kpd* ENT
27	1B	]	67	43	F9			
28	1C	RET	68	44	F10	116	74	*Left m/j but*
29	1D	CNTL	69	45	*n.u*	117	75	*Right m/j but*
30	1E	A	70	46	*n.u*			
31	1F	S	71	47	HOME			
32	20	D	72	48	↑			
33	21	F	73	49	*n.u*			
34	22	G	74	4A	*kpd* -			
35	23	H	75	4B	←			
36	24	J	76	4C	*n.u*			
37	25	K	77	4D	→			
38	26	L	78	4E	*kpd* +			
39	27	;	79	4F	*n.u*			
40	28	'	80	50	↓			

UK Keyboard

43	2B	#
96	60	\

kpd = keypad *n.u* = not used *m/j but* = mouse/joystick button

Returned highword lowbyte by BDOS c_conin function

GEM VDI STANDARD KEYBOARD CODES

Hi byte	*Lo byte*	*Char*	*Hi byte*	*Lo byte*	*Char*	*Hi byte*	*Lo byte*	*Char*
03	00	CTRL2	28	27	!	31	4E	N
1E	01	CTRLA	0A	28	(	18	4F	O
30	02	CTRLB	0B	29	)	19	50	P
2E	03	CTRLC	09	2A	*	10	51	Q
20	04	CTRLD	0D	2B	+	13	52	R
12	05	CTRLE	33	2C	,	1F	53	S
21	06	CTRLF	0C	2D	-	14	54	T
22	07	CTRLG	34	2E	.	16	55	U
23	08	CTRLH	35	2F	/	2F	56	V
17	09	CTRLI	0B	30	0	11	57	W
24	0A	CTRLJ	02	31	1	2D	58	X
25	0B	CTRLK	03	32	2	15	59	Y
26	0C	CTRLL	04	33	3	2C	5A	Z
32	0D	CTRLM	05	34	4	1A	5B	[
31	0E	CTRLN	06	35	5	2B	5C	\
18	0F	CTRLO	07	36	6	1B	5D	]
19	10	CTRLP	08	37	7	07	5E	^
10	11	CTRLQ	09	38	8	0C	5F	_ (u/score)
13	12	CTRLR	0A	39	9	29	60	'
1F	13	CTRLS	27	3A	:	1E	61	a
14	14	CTRLT	27	3B	;	30	62	b
16	15	CTRLU	33	3C	<	2E	63	c
2F	16	CTRLV	0D	3D	=	20	64	d
11	17	CTRLW	34	3E	>	12	65	e
2D	18	CTRLX	35	3F	?	21	66	f
15	19	CTRLY	03	40	@	22	67	g
2C	1A	CTRLZ	1E	41	A	23	68	h
1A	1B	CTRL[	30	42	B	17	69	i
2B	1C	CTRL\	2E	43	C	24	6A	j
1B	1D	CTRL]	20	44	D	25	6B	k
07	1E	CTRL6	12	45	E	26	6C	l
0C	1F	CTRL-	21	46	F	32	6D	m
39	20	SPACE	22	47	G	31	6E	n
02	21	!	23	48	H	18	6F	o
28	22	"	17	49	I	19	70	p
2B	23	#	24	4A	J	10	71	q
05	24	$	25	4B	K	13	72	r
06	25	%	26	4C	L	1F	73	s
08	26	&	32	4D	M	14	74	t

Continued . . .

GEM VDI standard keyboard codes – continued

Hi byte	*Lo byte*	*Char*	*Hi byte*	*Lo byte*	*Char*	*Hi byte*	*Lo byte*	*Char*
16	75	u	11	00	ALTW	71	00	* F40
2F	76	v	2D	00	ALTX	73	00	CTRL←
11	77	w	15	00	ALTY	4D	00	→
2D	78	x	2C	00	ALTZ	4D	36	SHIFT→
15	79	y	3B	00	F1	74	00	CTRL→
2C	7A	z	3C	00	F2	50	00	↓
1A	7B	{	3D	00	F3	50	32	SHIFT↓
2B	7C	|	3E	00	F4	48	00	↑
1B	7D	}	3F	00	F5	48	38	SHIFT↑
29	7E	~	40	00	F6	51	00	* PAGE DOWN
0E	7F	DEL	41	00	F7	51	33	* SHIFT PG DN
81	00	ALT0	42	00	F8			
78	00	ALT1	43	00	F9	76	00	* CTRL PG DN
79	00	ALT2	44	00	F10	49	00	* PAGE UP
7A	00	ALT3	54	00	SHIFT F1*	49	39	* SHIFT PG UP
7B	00	ALT4	55	00	SHIFT F2	84	00	* CTRL PG UP
7C	00	ALT5	56	00	SHIFT F3	77	00	CTRL HOME
7D	00	ALT6	57	00	SHIFT F4	47	00	HOME
7E	00	ALT7	58	00	SHIFT F5	47	37	SHIFT HOME
7F	00	ALT8	59	00	SHIFT F6	52	00	INSERT
80	00	ALT9	5A	00	SHIFT F7	52	30	SHIFT INSERT
1E	00	ALTA	5B	00	SHIFT F8	53	00	DELETE
30	00	ALTB	5C	00	SHIFT F9	53	2E	SHIFT DEL
2E	00	ALTC	5D	00	SHIFT F10	72	00	* CTRL PRINT SCREEN
20	00	ALTD	5E	00	*F21			
12	00	ALTE	5F	00	*F22			
21	00	ALTF	60	00	*F23	37	2A	* PRINT SCRN
22	00	ALTG	61	00	*F24	01	1B	ESCAPE
23	00	ALTH	62	00	*F25	0E	08	BACKSPACE
17	00	ALTI	63	00	*F26	82	00	ALT-
24	00	ALTJ	64	00	*F27	83	00	ALT=
25	00	ALTK	65	00	*F28	1C	0D	CR
26	00	ALTL	66	00	* F29	1C	0A	CTRL CR
32	00	ALTM	67	00	* F30	4C	35	SHIFT*kpd* 5
31	00	ALTN	68	00	* F31	4A	2B	*kpd* -
18	00	ALTO	69	00	* F32	4E	2B	*kpd* +
19	00	ALTP	6A	00	* F33	0F	09	TAB
10	00	ALTQ	6B	00	* F34	0F	00	* BACKTAB
13	00	ALTR	6C	00	* F35	4B	00	→
1F	00	ALTS	6D	00	* F36	4B	34	SHIFT→
14	00	ALTT	6E	00	* F37	4F	00	* END
16	00	ALTU	6F	00	* F38	4F	31	* SHIFT END
2F	00	ALTV	70	00	* F39	75	00	* CTRL END

* These scan codes are not supported by Atari ST BIOS

Appendix E – List of callable functions

GEM BIOS calls

Code Dec	Code Hex		Function (called from trap#13)
0	00	getmpb	Get & fill memory parameter block
1	01	bconstat	Return character-device input status
2	02	bconin	Input character to device, ret if done
3	03	bconout	Ooutput character to device, ret if done
4	04	rwabs	Rd/wr logical sectors to device
5	05	setexc	Get or set vector number
6	06	tickcal	Return system timer value (ms)
7	07	getbpb	Return pointer to BIOS param block
8	08	bcostat	Return character output device status
9	09	mediach	Check for media change
10	0A	drvmap	Set bit map & logical drives
11	0B	kbshft	Set keyboard shift bits

Callable from user mode
Re-entrant to three levels

Device = 0 Printer, parallel port
= 1 Aux, RS232 port
= 2 Con, screen
= 3 Midi
= 4 Keyboard

Extended ST BIOS calls

Code Dec	Code Hex		Function (called by trap#14)
0	00	inimous	Initialise mouse packet handler
1	01	ssbrk	Reserve X bytes from top memory
2	02	_physbase	Get screen's physical base address
3	03	_logbase	Get screen's logical base
4	04	_getrez	Get screen's current resolution
5	05	_setScreen	Set screen's logical location
6	06	_setPallete	Set contents of hardware pallette
7	07	_setColor	Set the pallette number
8	08	_floprd	Read sectors from floppy disk
9	09	_flopwr	Write sectors to floppy disk

Extended ST BIOS calls continued

Code Dec	Code Hex		Function (called by trap#14)
10	0A	_flopfmt	Format floppy disk
11	0B	getdsb	Get device status block pointer
12	0C	midiws	Write string to MIDI port
13	0D	_mfpint	Set MFP interrupt number
14	0E	iorec	Ret pointer to serial device buffer record
15	0F	rsconf	Configure RS232 port
16	10	keytbl	Set pointer to keyboard translation tabl
17	11	_random	Ret 24 bit pseudo random number
18	12	_protobt	Prototype image boot sector
19	13	_flopver	Verify sectors from floppy
20	14	scrdmp	Dump screen to printer
21	15	cursconf	Get/set cursor blink/attributes
22	16	settime	Set keyboard time & date
23	17	gettime	Get time & date from keyboard
24	18	bioskeys	Restore keyboard translation tables
25	19	ikbdws	Write string to interrupt keyboard
26	1A	jdisint	Disable interrupt no. on MK68901
27	1B	jenabint	Enable interrupt no. on MK68901
28	1C	giaccess	Read/write sound chip reg
29	1D	offgibit	Set port A bit 0 atomically
30	1E	ongibit	Set port A bit 1 atomically
31	1F	xbtimer	Set MFP timers & control registers
32	20	dosound	Set pointer to command bytes
33	21	setprt	Set/get printer configuration
34	22	kbdvbase	Ret pointer to keyboard structure
35	23	kbrate	Get/set keyboard repeat rate
36	24	_prtblk	prtblk primitive
37	25	vsync	Wait for next vblank
38	26	supexec	Execute in super mode
39	27	puntaes	Throw away AES

Callable from user mode.

GEM BDOS calls

Code Dec	Code Hex		Function (called by trap#1)
0	00	p_termo	Terminate process (use $4C)
1	01	c_conin	Read character from standard input
2	02	c_conout	Write character to standard output
3	03	c_auxin	Read character from aux device
4	04	c_auxout	Write character to aux device
5	05	c_prnout	Write character to standard print device
6	06	c_rawio	Raw I/O to standard I/O
7	07	c_rawcin	Raw input from standard input
8	08	c_necin	Read character standard input (no echo)
9	09	c_conws	Write null term $ standard input
10	0A	c_conrs	Read edited $ from standard input
11	0B	c_conis	Check status of standard input
14	0E	d_setdrv	Set default drive
16	10	c_conos	Check status of standard output
17	11	c_prnos	Check stat standard print device
18	12	c_auxis	Check stat standard aux device input
19	13	c_auxos	Check stat standard aux device output
25	19	d_getdrv	Get default drive
26	1A	f_setdta	Set disk transfer address
42	2A	t_getdate	Get date
43	2B	t_setdate	Set date
44	2C	t_gettime	Get time
45	2D	t_settime	Set time
47	2F	f_getdta	Get disk transfer address
48	30	s_version	Get version number
49	31	p_termres	Terminate and stay resident
54	36	d_free	Get drive free space
57	39	d_create	Create a subdirectory
58	3A	d_delete	Delete a subdirectory
59	3B	d_setpath	Set current directory
60	3C	f_create	Create a file
61	3D	f_open	Open file
62	3E	f_close	Close file
63	3F	f_read	Read file
64	40	f_write	Write file
65	41	f_delete	Delete file
66	42	f_seek	Seek file pointer
67	43	f_attrib	Get/Set file attribute
69	45	f_dup	Duplicate file handle
70	46	f_force	Force file handle
71	47	d_getpath	Get current directory

GEM BDOS calls continued

Code Dec	Code Hex		Function (called by trap#1)
72	48	m_alloc	Allocate memory
73	49	m_free	Free allocated memory
74	4A	m_shrink	Shrink size of allocated memory
75	4B	p_exec	Load or execute a process
76	4C	p_term	Terminate process
78	4E	f_sfirst	Search for 1st occurence of filspec
79	4F	f_snext	Search for 2nd occurence of filspec
86	56	f_rename	Rename a file
87	57	f_datime	Get/set file date & time stamp

Extended ST BDOS function calls

Code Dec	Code Hex	Function	(called by trap#2)
0	00	System reset	System/program control
115	73	VDI access	
200	C8	AES access	

GEM VDI functions

Op code		Definition	Output device Scrn	Prnt	Plot	Mfil
*1		Open workstation Use virtual	x	x	x	x
*2		Close workstation workstation	x	x	x	x
3		Clear workstation	x	x	x	x
4		Update workstation	x	x	x	x
5		*Escape code*				
	1	Inquire address character cells	x	x	x	x
	2	Exit alpha mode	x			x
	3	Enter alpha mode	x			x
	4	Cursor up	x			
	5	Cursor down	x			
	6	Cursor right	x			
	7	Cursor left	x			
	8	Home cursor	x			
	9	Erase to screen end	x			
	10	Erase to line end	x			
	11	Direct cursor address	x			
	12	Output cursor addressable text	x			
	13	Reverse video on	x			
	14	Reverse video off	x			
	15	Inquire curr. alpha cursor add	x			
	16	Inquire tablet status			x	
	17	Hard copy		x		
	18	Place graphic cursor	x			
	19	Remove last graphic cursor	x			
*	20	Form advance		x		x
*	21	Output window		x		x
*	22	Clear display list		x		x
*	23	Output bit image file		x		x
*	60	Select pallette				x
*	91	Inquire pallette film types				x
*	92	Inquire pallette driver state				x
*	93	Set pallette driver state				x
*	94	Save pallette driver state				x

* Not implemented on atari ST

Scrn = screen Prnt = printer
Plot = plotter Mfil = metafile

GEM VDI functions continued

Op code		Definition	Output device Scrn	Prnt	Plot	Mfil
5		*Escape codes*				
*	95	Supress pallette messages				x
*	96	Pallette error inquire				x
*	98	Update metafile extents				x
*	99	Write metafile item				x
*	100	Change GEM VDI filename				x
6		Polyline	x	x	x	x
7		Polymarker	x	x	x	x
8		Text	x	x	x	x
9		Filled area	x	x	x	x
10		Cell array	x	x	x	x
11	Esc code	Generalized Drawing Primitives				
	1	Bar	x	x	x	x
	2	Arc	x	x	x	x
	3	Pie	x	x	x	x
	4	Circle	x	x	x	x
	5	Ellipse	x	x	x	x
	6	Elliptical arc	x	x	x	x
	7	Elliptical pie	x	x	x	x
	8	Rounded rectangle	x	x	x	x
	9	Filled rounded rectangle	x	x	x	x
	10	Justified graphics text	x	x	x	x
12		Set char height abs. mode	x	x	x	x
13		Set character baseline vector	x			x
14		Set colour representation	x			x
15		Set polyline linetype	x	x	x	x
16		Set polyline line width	x			x
17		Set polyline colour index	x	x	x	x
18		Set polymaker type	x	x	x	x
19		Set polymarker height	x			x
20		Set polymarker colour index	x	x	x	x
21		Set text face	x	x	x	x

* Not implemented on atari ST

Scrn = screen Prnt = printer
Plot = plotter Mfil = metafile

GEM VDI functions continued

Op code	Definition	Output device Scrn	Prnt	Plot	Mfil
22	Set text colour index	x	x	x	x
23	Set fill interior style	x	x	x	x
27	Inquire cell array	x			x
*28	Input locator	x			x
*29	Input valuator, request/sample	x			x
*30	Input choice, request/sample	x			x
*31	Input string	x			x
32	Set writing mode	x	x		x
*33	Set input mode	x			x
35	Inquire curr polyline attributes	x	x	x	x
36	Inquire curr polymarker attr's	x	x	x	x
37	Inquire curr fill area attributes	x	x	x	x
38	Inquire curr graphic text attribs	x	x	x	x
39	Set graphic text alignment	x	x	x	x
100	Open virtual screen workstation	x			
101	Close virtual screen workstation	x			
102	Extended inquire function	x	x	x	x
103	Contour fill				x
104	Set fill perimeter visibility	x	x	x	x
105	Inquire pixel				
106	Set graph text special effects	x	x		x
107	Set char cell ht, pnts mode	x	x	x	x
108	Set polyline and styles	x	x	x	x
109	Copy raster, opaque	x			
110	Transform form	x			
111	Set mouse form	x			
112	Set user-defined fill pattern	x	x		x
113	Set user-defined linestyle	x			x
114	Fill rectangle	x			x
115	Inquire input mode	x			
116	Inquire text extent	x	x	x	
117	Inquire character cell width	x	x	x	x

* Not implemented on atari ST

Scrn = screen Prnt = printer
Plot = plotter Mfil = metafile

GEM VDI functions continued

Op code	Definition	Output device Scrn	Prnt	Plot	Mfil
118	Exchange timer interrupt vector	x			
119	Load fonts	x			
120	Unload fonts	x			
121	Copy raster, transparent	x			
122	Show cursor	x			
123	Hide cursor	x			
124	Sample mouse button state	x		x	
125	Exchange button change vector	x			
126	Exch. mouse movement vector	x			
127	Exchange cursor change vector	x			
128	Sample keyboard state info	x			
129	Set clipping rectangle	x	x		x
130	Inquire facename and index	x	x	x	
131	Inquire current face info	x	x	x	x

* Not implemented on atari ST

Scrn = screen Prnt = printer
Plot = plotter Mfil = metafile

The standard range of VDI function output devices include a camera and a tablet as well as the screen, printer, plotter and metafile. Only the screen is implemented on the Atari ST.

GEM AES function calls

Op code	Description	
Application library routines		
10	Initialise application	APPL_INIT
11	Read message from pipe	/APPL_READ
12	Write message to pipe	APPL_WRITE
13	Find another application	APPL_FIND
14	Playback GEM recording	APPL_TPLAY
15	Record GEM session	APPL_TRECORD
19	Cleanup and exit	APPL_EXIT
Timer event routines		
20	Waiting for keyboard input	EVNT_KEY
21	Waiting for button input	EVNT_BUTTON
22	Waiting for mouse input	EVNT_MOUSE
23	Waiting for messag input	EVNT_MESAG
24	Waiting period	EVNT_TIMER
25	Waitg for multi-events	EVNT_MULTI
26	Get/set mouse clickrate	EVNT_DCLICK
Menu library routines		
30	Toggle application menu bar	MENU_BAR
31	Toggle menu check mark	MENU_ICHECK
32	Toggle menu item able	MENU_IENABLE
33	Toggle display video	MENU_TNORMAL
34	Change item menu text	MENU_TEXT
35	Put acc's menu in desk	MENU_REGISTER
Object library routines		
40	Add object to tree	OBJC_ADD
41	Delete objct from tree	OBJC_DELETE
42	Draw an object or tree	OBJC_DRAW
43	Find objct under mouse	OBJC_FIND
44	Compute object offset	OBJC_OFFSET
45	Change obj tree order	OBJC_ORDER
46	Edit objects text	OBJC_EDIT
47	Change objects state	OBJC_CHANGE

GEM AES function calls – continued

Form library routines		
50	Monitor user/form	FORM_DO
51	Toggle dialog boxes	FORM_DIAL
52	Display alert box	FORM_ALERT
53	Display error box	FORM_ERROR
54	Centre dialog box	FORM_CENTER
Graphics library routines		
70	Draw a rubber box	GRAF_RUBBERBOX
71	Drag a box around	GRAF_DRAGBOX
72	Draw moving box	GRAF_MOVEBOX
73	Draw expanding outline	GRAF_GROWBOX
74	Draw shrinking outline	GRAF_SHRICKBOX
75	Test for mouse inside	GRAF_WATCHBOX
76	Slide box in parent	GRAF_SLIDEBOX
77	Return screen handle	GRAF_HANDLE
78	Redefine mouse form	GRAF_MOUSE
79	Return mouse attributes	GRAF_MKSTATE
Scrap library routines		
80	Read clipboard directory	SCRP_READ
81	Write directory to clipboard	SCRP_WRITE
File selector routines		
90	Display file selector box	FSEL_INPUT
Window library routines		
100	Allocate full window	WIND_CREATE
101	Open window to size	WIND_OPEN
102	Close window	WIND_CLOSE
103	Deallocate window	WIND_DELETE
104	Get window data	WIND_GET
105	Set window data	WIND_SET
106	Find mouse window	WIND_FIND
107	Update window	WIND_UPDATE
108	Calculate window data	WIND_CALC

GEM AES function calls – continued

Resource library routines

110	Load resource file	RSRC_LOAD
111	Deallocate resource file	RSRC_FREE
112	Get structure address	RSRC_GADDR
113	Save structure index	RSRC_SADDR
114	Convert characters to pixels	RSRC_OBFIX

Shell library routines

120	Find how created	SHELL_READ
121	Exit AES or run other	SHELL_WRITE
122	Get data	
123	Put data	
124	Find filename path	SHELL_FIND
125	Find parameter address	SHELL_ENVRN

Intelligent keyboard (ikbd) command set

Command Code		
Dec	*Hex*	*Function*
128	80	Reset return keyboard to power-up
1	01	status without affecting clock. A break > 200ms also causes a reset
7	07	Set mouse button action
8	08	Set mouse relative position reporting
9	09	Set mouse absolute positioning
10	0A	Set mouse keycode mode
11	0B	Set mouse threshold
12	0C	Set mouse scale
13	0D	Interrogate mouse position
14	0E	Load mouse position
15	0F	Set Y = 0 at bottom
16	10	Set Y = 0 at top
17	11	Resume
18	12	Disable mouse
19	13	Pause output
20	14	Set joystick event reporting
21	15	Set joystick interrogation mode
22	16	Joystick interrogation
23	17	Set joystick monitoring
24	18	Set fire button monitoring
25	19	Set joystick keycode mode
26	1A	Disable joysticks
27	1B	Set time of day clock
28	1C	Interrogate time of day clock
32	20	Memory load
33	21	Memory read
34	22	Controller execute

Status inquiries (OR 80H with command)

The status of the keyboard can be determined by interrogating the status register in the configuration tables.

Code Dec	Hex	Function		
246	F6	Status report		
247	F7	Absolte mouse position record		
248	F8	Relative mouse position record		
248	F8	111110xx (xx = right-left button state)		
	F9	delta x, 2's complement		
	FA	delta y, 2's complement		
252	FC	Time of day (resolution of 1 second)		
253	FD	Joystick report header (both sticks)		
254	FE	x000yyyy	x = trigger	Joystick 0 event
255	FF	x000yyyy	y = stck posn	Joystick 1 event

A-line routines

Dec	*Hex*	*'A' line function*
20480	A000	Initialization
20481	A001	Put pixel
20482	A002	Get pixel
20483	A003	Line
20484	A004	Horizontal line
20485	A005	Filled rectangle
20486	A006	Line-by-line filled polygon
20487	A007	BitBlt (including half tone source patterns)
20488	A008	TextBlt (all 16 BitBlt logic operations)
20489	A009	Show mouse
20490	A00A	Hide mouse
20491	A00B	Transform mouse
20492	A00C	Undraw sprite
20493	A00D	Draw sprite
20494	A00E	Copy raster form
20495	A00F	Seedfill

Appendix F
Parameter blocks

SYSTEM BLOCKS

System start-up block

offset				
	$00	Reseth	Branch to reset handler	
	$02	Vers	OS version number	
	$04	Reseth	System reset handler	
	$08	Ostext	Base of OS	Pointers
12	$0C	Endos	End of OS mem used	Pointers
16	$10	Reseth	Default shell	
20	$14	Magic	Verification no. or zero	
24	$18	Date	System build date	

Boot sector parameter block

offset			
	$00	BRA.S	Branch to boot code
	$02	OEM's space	Reserved for OEMs use
	$08	Vol ser #	24 bit volume serial number
11	$0B	BPS	Number of bytes/sector
13	$0D	SPCs	Number of sectors/cluster
14	$0E	RES	Number of reserved sectors
16	$10	NFATS	Number of file alocation tables
17	$11	NDIRS	Number of directory entries
19	$13	NSECTS	Number of sectors on media
21	$15	MEDIA	Media descriptor - not used
22	$16	SPF	Number of sectors/FAT
24	$18	SPT	Number of sectors/track
26	$1A	NSIDES	Number of sides on media
28	$1C	NHID	Number of hidden sectors-not used
30	$1E	boot code	Start of code, if any ?
511	$1FE	last word	Used for checksum
512	$200		

DEVICE DRIVERS

Each device has one driver (Device control block – DCB) that contains entry points to routines and constants used by systems to initialize the device's state during a warm-start. The routines and constants are defined as follows:

Device driver

offset			
	$00	BREAD	Read sector
	$04	BWRITE	Write sector
	$08	BINIT	Initialize drive (warm start)
12	$0C	BFORMAT	Format drive
16	$10	BINTR	Vblank call (timeout-homing)
20	$14	BRDTRK	Read track
24	$18	BWRTRK	Write track
28	$1C	BXLATE	Logical to physical translate
32	$20	BCVSIZ	CSV size allocation
34	$22	BALVSIZ	ALV size allocation
38	$26	BDEFINFO	Default info block
42	$2A		

Device drivers are stored in RAM in a device state block (DSB), the DSB contains TOS specific data structures (the DPB and DPH) and device specific information, such as the number of tracks, head seek rate. The DSB is allocated during a warm-start.

Device state block

offset			
	$00	DDPH	Device parameter header
26	$1A	DDPB	Disk parameter block
42	$2A	DINFOSIZ	DSB size (not incl DDPH)
44	$2C	DPHYSDEV	Device physical number
46	$2E	DNTRACKS	No. tracks on device
48	$30	DSPT	No. sectors/track
50	$32	DNSIDES	No. sides/device
52	$34	DSEEKRT	Floppy seek rate
54	$36		

Floppy parameter block

offset			
	$00	Flock	Floppy lock return address
	$04	Cret	Callers return address
	$08	Dmapn	DMA pointer
12	$0C	—	Obsolete
16	$10	Devno	Device number
18	$12	Secno	Sector number
20	$14	Trkno	Track number
22	$16	Sidno	Side number
24	$18	Secnt	Sector count
26	$1a		

Sector buffer block

offset			
	$00	BNEXT	Next buffer or null
	$04	BBUF	Size of buffer (512 bytes)
	$08	BLRU	LRU replacement value
12	$0C	BFLAGS	Valid/dirty flags
14	$0E	BDEV	Device number
16	$10	BTRACK	Track number
18	$12	BSIDE	Side number
20	$14	BSSECT	Start sector number
22	$16	BESECT	End sector number
24	$18	BPSECT	Physical sector number
26	$1A	BSIZE	

PROGRAM PARAMETER BLOCKS

Transient program area block

offset

	Low TPA
Base page	To maintain maximum GEM DOS compatibility, free unused memory and lower top of stack (4A).
Text	
Data	
BSS	Determine memory available and allocate it.
Application user area	High TPA

Load block

offset			
	$00		Opened program file address
	$04		Base address to load program
	$08		Program end address + 1
12	$0C		Address of base page
16	$10		Default user stack pointer
20	$14		Loader control flags
22	$16		0_load at bottom
			1_load at top (reserved)

Base page format block

offset			
	$00	Low TPA	Base address of TPA
	$04	Hi TPA	End of TPA + 1
	$08	Tbase	Base address of text
12	$0C	Tlen	Length of text
16	$10	Dbase	Base address of init data
20	$14	Dlen	Length of data
24	$18	Bbase	Base add of BSS uninit data
28	$1C	Blen	Length of BSS uninit data

Atari OS specific base page

offset			
32	$20	Length free memory after BSS	
36	$24	Drive from which prog loaded	
37	$25	Reserved by BDOS	
56	$38	2nd parsed FCB (Command line)	Set by CCP
92	$5C	1st parsed FCB (Command line)	Set by CCP
128	$80	Command tail and default	Set by CCP
		DMA buffer	
		$FF end	

GEMDOS specific base page

offset			
32	$20		DTA address pointer
36	$24		Parent's base page pointer
40	$28		Reserved
44	$2C	Environ	Environment string pointer
128	$80	Cmdline	Command line image

File header

offset			
			601AH data & BSS contiguous
	$00	BRA.S flag	else 601BH (not supported by ST OS)
	$02		Bytes in text segment
	$06		Bytes in data segment
10	$0A		Bytes in BSS
14	$0E		Bytes in symbol table
18	$12		Zero (reserved)
22	$16		Start of text seg & prog exec
26	$1A		Zero if no relocation bits

Extension to file header if BSS and data not contiguous:
(not supported by ST OS)

offset		
28	$1C	Start address of data segment
32	$20	Start address of BSS
36	$24	

Memory parameter block

offset		
$00	Owner description	Memory descriptor
	No. bytes in block	Memory descriptor
	Start address of block	Memory descriptor
	Next link MD	Memory descriptor
	Roving pointer	
	Memory allocation list	
	Memory free list	

GEM PARAMETER BLOCKS

VDI parameter block

offset			Longword address
	$00	contrl	Control table pointer
	$04	intin	input attribute table pointer
	$08	ptsin	input points table pointer
12	$0C	intout	output attribute table pointer
16	$10	ptsout	output points table pointer
20	$14		

VDI control table

offset					
	$00	Op code	Function op code		
	$02	Lptsin	input coordinate	Size in Wordpairs	Table sizes
	$04	Lptsout	output coordinate		
	$06	Lintin	input attribute	Size in words	
	$08	Lintout	output attribute		
10	$0A		Subfunction ident number		
12	$0C		Device handle		
14	$0E		Op code dependent information		

AES parameter block

offset			Longword address
	$00	control	Control table pointer
	$04	global	Global array pointer
	$08	int_in	input attribute table pointer
12	$0C	int_out	input points table pointer
16	$10	addr_in	output attribute table pointer
20	$14	addr_out	output points table pointer
24	$18		

AES control table

offset					
	$00	Op code	Function op code		
	$02	Lint in	input coordinate	Size in words	Table Sizes
	$04	Lint out	output coordinate		
	$06	Laddr in	input attribute	Size in longwords	
	$08	Laddr out	output attribute		
10	$0A				

AES global array

offset			
	$00	version	GEM AES version ident word
	$02	count	Max #concurrent applics allowed
	$04	id	Unique application identifier
	$06	private	Longword user data
10	$0A	ptree	Resource address tree pointer
14	$0E	reserved	Zero
18	$12	reserved	Zero
22	$16	reserved	Zero
26	$1A	reserved	Zero
30	$1E		

A-LINE VARIABLES

A-line routine table

offset		*Function*
$00	0	Number of video planes } Can produce special
$02	2	Number of bytes/video line } effects.
$04	4	Pointer to Cntrl array
$08	8	Pointer to Intin array
$0C	12	Pointer to Ptsin array
$10	16	Pointer to Intout array
$14	20	Pointer to Ptsout array
$18	24	Bit plane 0 } current
$1A	26	Bit plane 1 } color
$1C	28	Bit plane 2 } value
$1E	30	Bit plane 3 }
$20	32	–1
$22	34	VDI line style equivalent
$24	36	Writing mode: 0 = replace; 1 = transparent 2 = XOR mode 3 = inverse transparent
$26	38	X1 coordinate
$28	40	Y1 coordinate
$2A	42	X2 coordinate
$2C	44	Y2 coordinate
$2E	46	Pointer to current fill pattern
$32	50	Fill pattern mask
$34	52	Multi-plane fill pattern 0_current fill pattern is single plane 1_current fill pattern is multi-plane
$36	54	Clipping flag: 0 = no clipping
$38	56	Minimum x clipping value
$3A	58	Minimum y clipping value
$3C	60	Maximum x clipping value
$3E	62	Maximum y clipping value
$40	64	Accumulator for textblt x dda initialize to 8000H before each call
$42	66	Textblt scale factor
$44	68	Scale direction 0_down
$46	70	Font status 1 = current font monospaced & may be thickened 0 = may not be thickened to increase font width
$48	72	X coor of character in font form
$4A	74	Y coor of character in font form (typically 0)
$4C	76	X coor of character on screen

Continued . . .

$4E	78	Y coor of character on screen
$50	80	Character width
$52	82	Character height
$54	84	Pointer to start of font data (font form)
$58	88	Width of font form
$5A	90	Style bit: 0 = Thicken, bit 1 = lighten, bit 2 = skew bit 3 = underline (ignored), bit 4 = outline
$5C	92	Lighten text mask
$5E	94	Skew text mask
$60	96	Text thickening additional width
$62	98	Offset above character baseline for skew
$64	100	Offset below character baseline for skew
$66	102	Scaling flag: 0 = no scaling
$68	104	Character rotation vector 0 = horizontal 900 = vert down, etc
$74	116	Copy raster form type flag (RAM VDI only) 0 = opaque type n-plane source to n-plane destination bitblt write mode <>0_transparent type 1-plane source to n-plane destination VDI write mode
$76	118	Abort fill routine pointer *(not available on disk based versions of TOS)*

The A-line variable table contains other parameters which may be of use to the programmer.

offset		*Function*
$D2	–46	Pixel cell height. (Same as font form's height)
$D4	–44	Maximum number of cells across –1 (X)
$D6	–42	Maximum number of cells high –1 (Y)
$D8	–40	Byte offset next vert cell. Scn wid(byte) × Pixel cell ht
$DA	–38	Physical color index of background color.
$DC	–36	Physical color index of foreground color.
$DE	–34	Current cursor address
$E2	–30	Byte offset from screen base to top of first cell
$E4	–28	Cursor position: cell x
$E6	–26	Cursor position: cell y
$E8	–24	Cursor flash interval (in frames)

Continued . . .

$E9	–23	Cursor countdown timer
$EA	–22	Address of monospace font data Each cell is 8 pixels wide and byte aligned. The data format is defined in the VDI manual. The cells may be arbitrarily high.
$EE	–18	Last ascii code in font
$F0	–16	First ascii code in font
$F2	–14	Width of font form in bytes
$F4	–12	Maximum x pixel value
$F6	–10	Address of font offset table (per VDI spec)
$FA	–06	Alpha text status byte bit 0 cursor flash 0 = disabled, 1 = enabled bit 1 flash state 0 = off, 1 = on bit 2 cursor visibility 0 = invisible, 1 = visible bit 3 end of line 0 = overwrite, 1 = wrap bit 4 reverse video 0 = on, 1 = off bit 5 cursor position saved 0 = false, 1 = true
$FC	–04	Maximum y pixel value of the screen

Sprite definition block

offset			
	$00	X offset of hot-spot	
	$02	Y offset of hot-spot	
	$04	Format flag	
	$06	Background	Color table index
	$08	Foreground	Color table index
10	$0A	Interleaved background/foreground	B'gnd line 0
12	$0C		F'gnd line 0
74	$4A	image (32 words)	F'gnd line 16
76	$4C		

Format flag

+ve Fg	*+ve* Bg	*–ve* Fg	*–ve* Bg	*color plotted*
0	0	0	0	Transparent
0	1	0	1	Background
1	1	1	1	Foreground
1	0			Foreground
		1	0	XOR screen

Memory form definition block (MFDB)

offset			
	$00	Mem pointer	32-bit address of pixel 0,0
	$04	Width	Raster area dimensions
	$08	Height	Raster area dimensions
12	$0C	Word width	Pixel width/word size
16	$10	Format flag	1 = standard, 0 = device specific
20	$14	Mem planes	No. planes in raster area
24	$18		Three reserved words
28	$1C		Three reserved words
32	$20		Three reserved words
36	$24		Three reserved words

HEADER BLOCKS

Cartridge header block

offset		*Prefix to application header*
252	$FC	Flag #$ABCDEF42 program/data or #$FA52255F diagnostic

Application header block

offset			
	$00	Next	Link to next application
	$04	Flag/init	Pointer to initialize code or run flag (MSB)
	$08	Run	Pointer to run code
12	$0C	Time	DOS-format } Time/Date
14	$0E	Date	DOS-format } application created
16	$10	Size	Application size
20	$14	Name	Application name (NNNNNNNN.EEE)

Run flag bit set:

Bit 0 Run before interrupt vectors and memory initialized
Bit 1 Run before GEMDOS initialized
Bit 2 unused
Bit 3 Run before disk boot
Bit 4 unused
Bit 5 Application is a desk accessory
Bit 6 Not a GEM application. No AES calls
Bit 7 Requires command line parameters before execution

Appendix G – MC68000 Instruction summary

ADDRESS MODE

Assembler language and BASIC equivalents

Address mode		*Source*	*Destination*
Data register direct	Dn	MOVE.L D2,D0 LET D0=D2	MOVE.L #999,D0 LET D0=999
Address register direct	An	MOVE.L A0,D0 LET D0=A0	MOVEA.L #999,A0 LET A0=999
Address register indirect	(An)	MOVE.L (A0),D0 LET D0,PEEK_L(A0)	MOVE.L #999,(A0) POKE_L (A0),999
Address register indirect with postincrement	(An)+	MOVE.L (A0)+,D0 LET D0,PEEK_L(A0) LET A0=A0 + 4	MOVE.L #999,(A0)+ POKE_L (A0),999 LET A0=A0 + 4
Address register indirect with predecrement	–(An)	MOVE.L -(A0),D0 LET A0=A0 - 4 LET D0,PEEK_L(A0)	MOVE.L #999, -(A0) LET A0=A0 - 4 POKE_L (A0),999
Address register indirect with displacement	d(An)	MOVE.L 9(A0),D0 LET D0=PEEK_L(9 + A0)	MOVE.L #999,9(A0) POKE_L(A0+9),999
Address register indirect with index	d(An.Ri)	MOVE.L 9(A0.D2),D0 LET D0=PEEK_L(9+A0+D2)	MOVE.L #999,9(A0.D0) POKE_L(A0+9+D0),999
Absolute short ABS.S	$xxxx	MOVE.L 1024,D0 LET D0=PEEK_L(1024)	MOVE.L #999,1024 POKE_L(1024),999
Absolute long ABS.L	$xxxxxx	MOVE.L 163840,D0 LET D0=PEEK_L(163840)	MOVE.L #999,163840 POKE_L(163840),999
Program counter with displacement	d(PC)	MOVE.L 9(PC),D0 LET D0=9 + conts of Program Counter	Not legal
Program counter with index	d(PC.Ri)	MOVE.L 9(PC.D2),D0 LET D0=9+D2 + cont of Program Counter	Not legal

ALLOWABLE ADDRESS MODE TYPES

	All	*Alt Mem Add*	*Data Alt'ble Addr'g*	*Alt Add Mod*	*Dat Add Md1*	*Dat Add Md2*	*Con Add Md1*	*Con Alt Add*	*Con Add Md2*	
	Source	*Dest.*	*Destination*		*Dest.*	*Source*	*Dest.*		*Dest.*	*Src.*
Dn	x		x	x	x	x	x			
An	x				x					
(An)	x	x	x	x	x	x	x	x	x	x
(An)+	x	x	x	x	x	x	x			x
–(An)	x	x	x	x	x	x	x		x	
d(An)	x	x	x	x	x	x	x	x	x	x
d(An.Ri)	x	x	x	x	x	x	x	x	x	x
ABS sh't	x	x	x	x	x	x	x	x	x	x
ABS lg	x	x	x	x	x	x	x	x	x	x
d(PC)	x					x	x	x		x
d(PC.Ri)	x					x	x	x		x
Imm	x					x				
	ADD	ADD	ADDI	NBCD	ADDQ	AND	BTST	JMP	MOVEM	MOVEM
	ADDA CMP CMPA MOVE MOVEA SUB SUBA ROXR ROL ROR LSL LSR	AND OR SUB ASL ASR ROXL EORI MOVE MOVE fr SR	ANDI BCHG BCLR BSET CLR CMPI EOR SUBI TAS TST	NEG NEGX NOT ORI Scc	SUBQ to SR MULS MULU OR	CHK DIVS DIVU MOVE to CCR MOVE		JSR LEA PEA	reg to mem	mem to reg

Alt = Alterable Mod = Mode Md1 = Mode1
Mem = Memory Dat = Data Md2 = Mode2
Add = Address Con = Control

(Types of addressing mode definitions used by Motorola to describe allowable modes)

Address modes

ENCODING

The range of addressing modes are coded consistently throughout the MC68000 instruction set and may be summarized as follows:

/table

Addressing mode	Syntax	Mode No.	Reg No.	Ext. words
Data register direct	Dn	0	n	0
Address register direct	An	1	n	0
Address register indirect	(An)	2	n	0
Address register indirect with postincrement	(An)+	3	n	0
Address register indirect with predecrement	–(An)	4	n	0
Address register indirect with displacement	d(An)	5	n	1
Address register indirect with index	d(An.Ri)	6	n	1
Absolute short ABS.S	$xxxx	7	0	1
Absolute long ABS.L	$xxxxxx	7	1	2
Program counter with displacement	d(PC)	7	2	1
Program counter with index	d(PC.Ri)	7	3	1
Immediate	#$xxx	7	4	1 or 2

n = Register number 0 to 7

Ext. Word = Number of extension words following the op word due to this address mode (source and destination ext. words are cumulative)

Mode No. = Dmod and Smod in instruction code tables

Reg No. = Dreg and Sreg in instruction code tables

Assembler language and BASIC equivalents continued

Address mode	*Source*	*Destination*

Immediate
Imm
	#$xxx
	`MOVE.L #65536,D0`
	`LET D0=65536`
		Not legal

Notes	Register D0 is used for the destination as an example; any other valid effective address may be used.	The source is defined as immediate data value 999; any other valid effective address may be used.

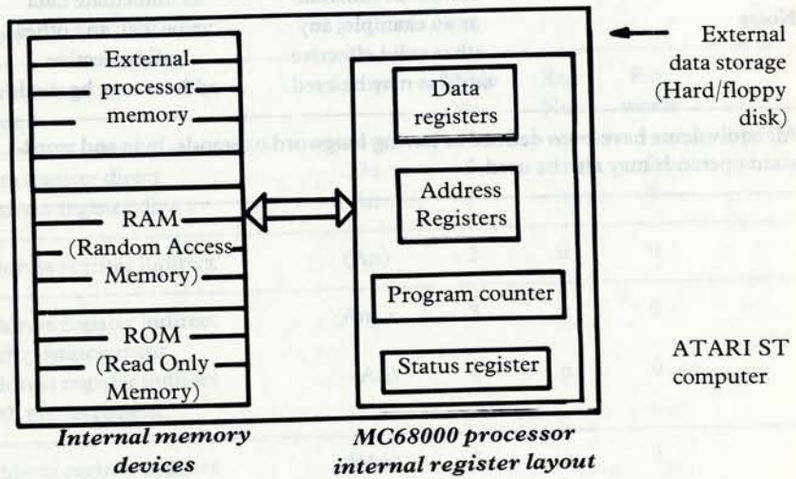
All equivalents have been defined as having longword operands, byte and word-sized operands may also be used.

Data storage

The MC68000 accesses two internal locations for storage:

Internal registers, of which there are 17, store the data inside the microprocessor itself. They are very limited in the amount of data they can store, but provide extremely fast access.

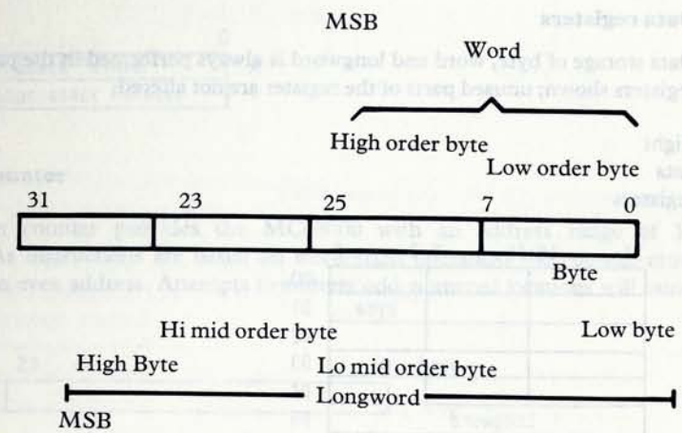
ST RAM/ROM, where data access is still quick, but not as fast as the internal register data access.



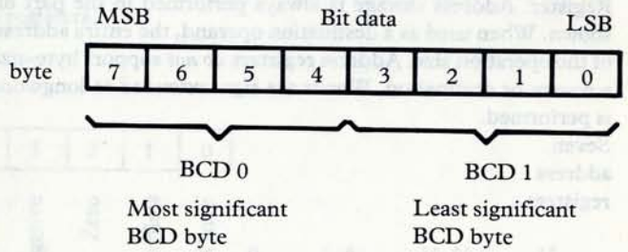
Data types

The MC68000 microprocessor supports five different data types; some instructions are limited to a specific data type, bust mostly there is an allowable range with the default of *word*. Where the choice is not implicit, it is defined in the instruction word extension as either *yte*, *word* or *longword*.

Byte, Word and Longword data types



BCD and IT data types



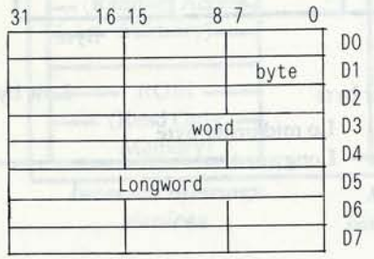
Internal registers

The Motorola 68000 has seventeen 32-bit registers, a 24-bit program counter and a 16-bit status register. eight of the 32-registers (D0 to D7) are used as data registersfor operations involving single-bit, bcd (4-bit), byte (8-bit), word (16-bit) and longword (32-bbit) data. The remaining nine registers are split into two; seven of these (A0 to A6) act as address registers, and two act as stack pointers. Only one stack pointer may be accessed at a ti,me, hence the convention of calling oth A7. The address register operations are based on word and longwords only.

Data registers

Data storage of byte, word and longword is always performed in the part of the data registers shown; unused parts of the register are not altered.

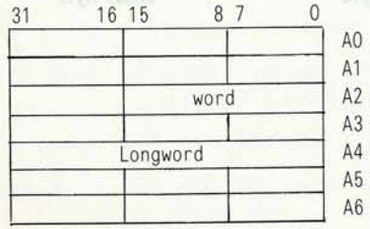
Eight
data
registers



Address registers

The address registers are used as pointers to user stacks, as base address registers and temporary storage for computed addresses tyhat are not to affect the Status Register. Address storage is always performed in the part of the address registers shown. When used as a destination operand, the entire address is changed regardless of the operation size. Address registers *do not* support byte-sized operations as either a source or destination. Words are sign-extended to longwords before an operation is performed.

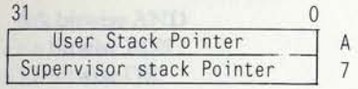
Seven
address
registers



Stack Pointer

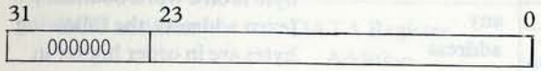
The user stack pointer typically saves subroutine returns when in user mode. The supervisor stack pointer saves the status register contents during trap and interrupt routines as well as the supervisor subroutine returns. Only one of the stack pointers is addressable at a time, so they are both called A7. ytes pushed on a stack are stored in the high order half of a word.

Two Stack
Pointers

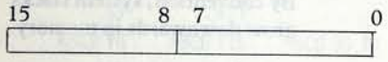


Program Counter

The program counter provides the MC68000 with an address range of 16 Megabytes. As instructions are based on word-sized operands, the counter must always hold an even address. Attempts to address odd-numered locations will cause an error trap.

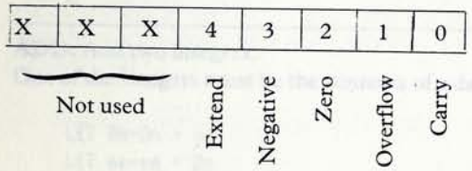


Status Register



The status register is split into *user* and *system* bytes. The user byte is evaluated for the condition codes used in the branching instructions. The codes are affected by all instructions that alter the contents of the data registers or memory, but not by changes to the address registers.

USER BYTE
Condition codes



The unused bits in the status register are read as zero. They are available for the MC68000 instruction set.

SYSTEM BYTE

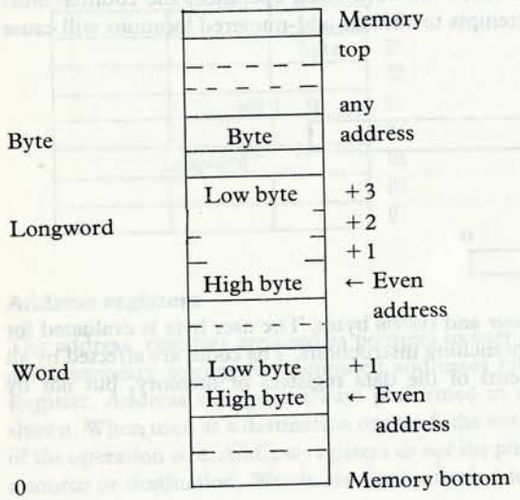
15	X	13	X	X	10	9	8

Trace mode

Supervisor state

Interrupt mask

Organisation of addresses in memory



For word and longword memory operations, the high byte is on a word boundary (even address), the following bytes are in order higher in memory.

By convention, system stacks grow downwards in memory.

INSTRUCTION SUMMARY

Each Motorola MC68000 instruction is presented, many in terms of equivalent BASIC Instructions or assembler routines. The similes are for clarification of the use of each instruction; there is no access to the data or address registers (Dn or An respectively) or the condition codes from BASIC and therefore the examples which make use of these registers, and most of the effective address modes (ea), cannot be taken literally.

Key

&& bitwise AND
^^ bitwise EOR
|| bitwise OR

ABCD: Add Binary Coded Decimal with Extend
Add two byte-sized binary coded decimal numbers and the Extend bit; a dollar sign is used to indicate a BCD number. Clear the extend bit and set the zero bit before performing this instruction which is limited to byte-size data register operations; multibyte additions are performed more easily in memory.

BCD addition	DATA Register Addition Byte only		Memory Multibyte Addition
			MOVE #4,CCR
$7	$27		ABCD -(A0),-(A1)
ABCD $6	ABCD $16	ABCD D0,D1	ABCD -(A0),-(A1)
$13	$43		ABCD -(A0),-(A1)

Note that the z-flag is cleared if the result is non-zero, otherwise it is unchanged and that in memory additions the data must be stored with the most significant digit lower in memory and the address pointers initially set to the byte above the low order BCD digit in memory, as the only available addressing mode is predecrement.

ADD: Add two integers.
One of the integers must be the contents of a data register.

```
LET Dn=Dn + ea                    ADD ea,Dn
LET ea=ea + Dn                    ADD Dn,ea
```

Use ADD ea,Dn where the destination is a data register.
Use ADDA where the destination is an address register.
Use ADDI or ADDQ where the source is immediate data.

Instruction summary – continued

ADDA: Add the contents of the effective address to the contents of the destination address register.

```
LET An=An + ea                    ADDA ea,An
```

ADDI: Add a constant value to the contents of the destination effective address. Use ADDQ for speed and small integers.

```
LET ea=ea + 999                   ADDI #999,ea
```

ADDQ: Add a constant in the range of 1 to 8 to the contents of the effective address. Faster addition than ADDI.

```
LET ea=ea + 8                     ADDQ #8,ea
```

ADDX: Add either register to register, or predecremented memory to memory, with extend. Use of the extend bit enables multiprecision arithmetic to be performed, the extend bit acting as a carry between successive operations.

Memory additions

```
ADDX -(Ay),-(Ax)
```

Where X infers the Extend bit

```
LET Ay=Ay-4
LET Ax=Ax-4
POKE(Ax),PEEK(Ax) + PEEK(Ay) + X
```

Data register addition
Add two 64 bit integers

```
D0_D1 and D2_D3 Lo–Hi resply
ADD.L D0,D2 Low bits
ADDX.L D1,D3 High bits
```

Memory addition

```
MOVE    #4,CCR
ADDX.L  -(A0),-(A1)
ADDX.L  -(A0),-(A1) etc.
```

Note that the z-flag is cleared if the result is non-zero, otherwise it is unchanged. For memory additions first clear the Extend bit and set the Zero flag. The data must be stored with the most significant digit lower in memory and the address pointers initially set the operand size above the low order digit in as the only addressing mode is predecrement.

Instruction summary – continued

AND: AND the source operand to the destination operand.
The source AND data is normally used either (a) as a mask enabling a portion of the destination operand to be examined (bits are masked by 1's in the source); or (b) to clear bits by setting the corresponding bit in the source to a zero.

```
LET ea=Dn && ea                   AND Dn,ea
LET Dn=src && Dn                  AND ea,Dn
```

If src = 3, then AND src keeps bits 0 and 1 in Dn only, the others are set to zero.

Use AND ea,Dn where the destination is a data register.
Use ANDA where the destination is an address register.
Use ANDI where the source is immediate data.

ANDI: ANDI the immediate data to the destination effective address.

```
LET ea=data && ea                 ANDI.W #512,D0
```

Keep bit 9 of word only

ANDI to CCR: ANDI the data to the condition code register.

```
LET CCR=26 && CCR                 ANDI #26,CCR
```

Normally bits can be tested via the condition codes without using the AND function as a mask. Here it is used to zero a bit position where there is a zero in the AND data; that is zero and carry (bits 0 and 2 in the CCR) are cleared.

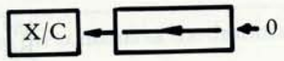
ANDI to SR: ANDI the data to the status register.
This is a privileged instruction and attempted access while in user mode will trap to the privilege violation exception vector.

```
LET SR=63743 && SR                ANDI #63743,SR
```

Set the interrupt mask level to zero and leave unchanged the condition code and system flags.

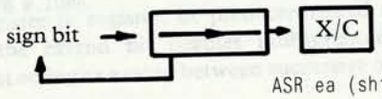
Instruction summary – continued

ASL: Arithmetically Shift Left the bits of the operand.
The last MSB shifted sets the carry and extend bits; the LSB is set to zero each shift. The overflow bit is set if the sign is changed during the shift and is used to flag a change of sign. The instruction is used for fast multiplication of ×2 and ×4; other values should use MULS.




```
LET ea=ea * 2                    ASL ea (shift 1)
LET Dy=Dy * (2∧Dx)               ASL Dx,Dy (reg modulo 64)
LET Dy=Dy * (2∧5)                ASL #5,Dy (shift 1 to 8)
```

ASR: Arithmetically Shift Right the bits of the operand.
The MSB sign bit is retained; the last LSB shifted is used to set the carry and extend bits. This instruction can be used for rapid integer division by 2, 4, 8 of signed numbers; use DIVS for other divisions.




```
LET ea=INT(ea/2)                 ASR ea (shift 1)
LET Dy=INT(Dy/(2∧Dx)             ASR Dx,Dy (reg modulo 64)
LET Dy=INT(Dy/(2∧5)              ASR #5,Dy (shift 1 to 8)
```

Bcc: Branch on condition a two's complement displacement from the current program counter position (Instruction address +2) +126 to –128 for a short branch or +32766 to –32768 for a word branch operation, the condition xx may be:

					2's complement arithmetic	
EQ	Equal To	CS	Carry Set		GT	Greater Than
NE	Not Equal	CC	Carry Clear		LT	Less Than
MI	Minus	VS	Overflow		GE	Greater Than or Equal to
PL	Plus	VC	No Overflow			
HI	Higher Than	LE	Less Than or Equal to			
LS	Lower Than or same					

```
IF Dn=0 THEN GOTO yy             BEQ #14
IF Dn>0 THEN GOTO label          BGT label
```


Instruction summary – continued

BCHG: A bit is tested and its state reversed.
If the bit was zero before the test; that is clear, then the Zero flag is set, otherwise it is cleared.

```
IF BITn = 0 THEN set_Zflag:
          ELSE clear_Zflag       BCHG #6,ea (data modulo 8)
LET BITn = 1-BITn                BCHG Dn,ea (reg modulo 32)
```

BCLR: A bit is tested and then cleared. If the bit was zero before the test; that is clear, then the Zero flag is set, otherwise it is cleared.

```
IF BITn = 0 THEN set_Zflag:
          ELSE clear_Zflag       BCLR #6,ea (data modulo 8)
LET BITn = 0                     BCLR Dn,ea (reg modulo 32)
```

BRA: BRanch Always.
A two's complement displacement branch either of +126 to –128 bytes by a single word instruction or of +32766 to –32768 bytes by a two-word instruction from the current program counter position (instruction address + 2).

```
GOTO label                       BRA label
GOTO 1275                        BRA #8
```

BSET: A bit is tested and then set.
If the bit was zero before the test; that is clear, then the Zero flag is set, otherwise it is cleared.

```
IF BITn = 0 THEN set_Zflag:
          ELSE clear_Zflag       BSET #6,ea (data modulo 8)
LET BITn=1                       BSET Dn,ea (reg modulo 32)
```

BSR: Branch to SubRoutine.
Either a two's complement displacement of +126 to –128 bytes by a single-word instruction, or of +32766 to –32768 bytes by a two-word instruction, from the current program counter position (instruction address +2). Return to the next instruction via an RTS from the subroutine.

```
GOSUB label                      BSR label
GOSUB 1275                       BSR #8
```

Instruction summary – continued

BTST: A bit is tested. If the bit was zero; that is clear, then the Zero flag is set, otherwise the Zero flag is cleared.

```
IF BITn = 0 THEN set_Zflag:          BTST #6,ea (data modulo 8)
      ELSE clear_Zflag               BTST Dn,ea (reg modulo 32)
```

CHK: Check a data register low-order word against the two's complement upper bound of the source operand.
If the register value is less than zero or greater than the test value, then jump to the CHK Trap exception vector.

```
IF Dn > ea OR                        CHK ea,Dn
   Dn < 0 THEN GOSUB chk_trap
```

CLR: Clear an operand sets all or part of a specified address or register to zero.

```
LET ea=0                             CLR ea
```

MOVEQ #0,Dn is quicker than CLR.L Dn

SUBA.L An,An is quicker for memory applications

CMP: The compare instructions are used exclusively to set the condition code registers for a subsequent conditional operation. The comparison is made by subtracting the source operand from the destination operand and setting the condition codes accordingly; neither operand is altered by the instruction.

```
                                     CMP ea,Dn
IF ea=Dn THEN GOTO loop              BEQ loop
```

Use CMPA when the destination is an address register.

Use CMPI when the source is immediate data.

Use CMPM for memory to memory comparisons.

CMPA: Subtract the source operand from the address register and set the condition code flags accordingly.
The comparison is based on a sign-extended source if it is a word operand. The address register is not altered.

```
                                     CMPA ea,Dn
```


Instruction summary – continued

CMPI: Subtract the immediate operand from the effective address operand and set the condition code flags accordingly.
Neither operand is altered. Use TST for comparing with zero as it is much quicker.

```
                                     CMPI #999,ea
```

CMPM: Subtract the contents of the memory address pointed to by the source address register from the contents of the memory address pointed to by the destination register and set the condition code flags accordingly. Increase the value of both address registers by the size of the operand (1, 2 or 4 byte word and longword respectively).

The main use for this instruction is comparing strings

```
LET Dn = length_string -1
loop                                 loop   CMPM (Ay)+,(Ax)+
IF PEEK (Ay) <> PEEK (Ax) THEN               BNE not_same
  Ay = Ay + s : Ax = Ax + s                  DBRA Dn,loop
  GOTO not_same                      next    .
 ELSE                                         .
  Ay = Ay + s:Ax = Ax + s                     .
  LET Dn = Dn -1                              .
  IF Dn = -1 THEN GOTO next                   .
  GOTO loop
END IF                               not_same .
next                                          .
 .                                            .
 .                                            .
not_same                                      Dn is the
 .        s = operand size                    character count
```

DBcc: Test the condition and exit loop to the next instruction if the condition is met.
If the condition is not met, then decrement the low order 16 bits of the count data register. If the count becomes –1, then exit loop and carry on with the next instruction, otherwise branch the two's complement displacement of the following word –32766 to +32768 from the current program counter position (instruction address +2). The test may be one of the following:

Instruction summary – continued

			2's complement arithmetic
EQ Equal To	CS Carry Set	GT Greater than	
NE Not Equal	CC Carry Clear	LT Less than	
MI Minus	VS Overflow	GE Greater than or Equal to	
PL Plus	VC No Overflow	LE Less than or Equal to	
HI Higher than	T True		
LS Lower than or same	F False		

```
DBEQ D0,loop        =    BEQ pass
                         SUB #1,D0
(Equivalent)             BPL loop
                     pass
```

DBT: Always branches and is of little use.

DBRA: Sometimes written DBF, it makes the branch based on the data register count only and branches when the count reaches –1. Therefore the count should be initialised to the required count –1. If the loop is entered via a jump or branch at the DBcc instruction, then the count is the required count and usefully an initial zero count will cause an immediate exit from the loop.

DIVS: Sign Divide a 32-bit data register destination operand by a 16 bit source operand and store the integer result in the lower 16 bits of the destination register. The remainder is stored in the upper 16 bits of the destination and keeps the dividend sign. Division by zero causes a jump to the Divide-by-Zero Trap exception vector. On overflow, the result is larger than 16 bits, the V-flag is set and the operation terminated without affecting either operand.

```
LET Dn=Dn / ea                DIVS ea,Dn

ASR   ea        is a fast signed divide by two.

MOVEQ #2,D2
ASR   D2,Dx     is a quicker divide by four.
```

Generally use DIVS and DIVU for division by a prime number, otherwise think of an alternative as the division instruction, because of its general nature, is not quick.

Instruction summary – continued

DIVU: Unsigned arithmetic divide of a 32-bit data register destination operand by a 16-bit source operand.
The integer result is stored in the lower 16 bits of the destination register and the remainder in the upper 16 bits. Division by zero causes a jump to the Divide-By-Zero exception vector. On overflow the result is larger than 16 bits, the V-flag is set and the operation terminated without affecting either operand.

```
LET Dn=Dn / ea                DIVU ea,Dn
```

EOR: EOR the data register source operand to the contents of the destination operand.
The source EOR data is normally used to invert the state of a bit or bits.

```
LET ea=Dn ^^ ea               EOR Dn,ea
```

If Dn = 3, then bits 0 and 1 in the effective address are inverted.

Use EORI where the source is immediate data. There is no memory to data register operation.

EORI: EORI the immediate data to the destination effective address.

```
LET ea=data ^^ ea             EORI.B #16,D0
```

Invert bit 4 of D0

EORI to CCR: EORI the immediate data to the condition code register.

```
LET CCR=4 ^^ CCR              EORI #4,CCR
```

Toggle the Zero_flag

EORI to SR: EORI the immediate data to the status register.
This is a privileged instruction and attempted access while in user mode will cause a trap to the privilege violation exception vector.

```
LET SR=8192 ^^ SR             EORI #8192,SR
```

Toggle the supervisor bit

Instruction summary – continued

EXG: Exchange the longword contents of two registers.
Referred to in many BASICs as SWAP, which has a different meaning in the MC68000 instruction code.

```
LET tmp=D0:D0=D1:D1=tmp        EXG D0,D1
LET tmp=A0:A0=A1:A1=tmp        EXG A0,A1
LET tmp=D0:D0=A0:A0=tmp        EXG D0,A0
```

EXT: Sign-extend a data register contents, a byte to a word or a word to a longword, to permit operations involving mixed size data to take place.

```
                               EXT Dn
```

ILLEGAL: The illegal instruction causes the processor to jump to the illegal instruction trap exception process subroutine.

```
GOSUB Ill_Trap                 ILLEGAL
```

JMP and **JSR:** JMP and JSR are long forms of BRA and BSR.
The main difference lies in the jump instruction's ability to access any part of memory whereas the branch instructions are limited to a relative ± 32K bytes jump.

JMP: Jump to a routine in memory specified by the effective address, either absolute or relative to the current program counter position.

```
GOTO ea                        JMP ea
```

JSR: Jump to a subroutine in memory specified by the effective address, either absolute or relative to the current program counter position

```
GOSUB ea                       JSR ea
```

LEA: Load Effective Address loads a calculated effective address into an address register.
The calculated address can be the sum of two registers, one must be an address register, and a displacement which provides the addition of two registers and a displacement without affecting either register, all in a single instruction.

```
Let An=Start_of_text address   LEA text,An
Let An=Start_of_table          LEA tabl,An
GOTO An + D0                   JMP 0(An.D0)

LET A0=A1 + D2 +64             LEA 64(A1.D2),A0
```


Instruction summary – continued

LINK: LINK enables a block of memory, part of the stack, to be temporarily reserved for a specific purpose; that is an index table, a text string, an array etc. and the space recovered when the requirement has passed.

```
DIM A(64)                      LINK An,# 64
```

Saves a block of 64 bytes in memory.
The original value of An is preserved on the stack and will be recovered on UNLK. The current value of An is the start of the data space which may be most easily accessed via indirect with displacement or indirect with index addressing modes.

LSL: Logically Shift Left the bits of the operand. The MSB sets the carry and extend bits, the LSB is set to zero.


```
LET ea=ea * 2                  LSL ea (shift 1)
LET Dy=Dy * (2∧Dx)             LSL Dx,Dy (reg modulo 64)
LET Dy=Dy * (2∧5)              LSL #5,Dy (shift 1 to 8)
```

LSR: Logically Shift Right the bits of the operand.
The MSB is set to zero and the LSB sets the carry and extend bits.


```
LET ea=INT(ea/2)               LSR ea (shift 1)
LET Dy=INT(Dy/(2∧Dx)           LSR Dx,Dy (reg modulo 64)
LET Dy=INT(Dy/(2∧5)            LSR #5,Dy (shift 1 to 8)
```

MOVE: Move the byte, word or longword contents of the source effective address to the destination effective address.

```
                               MOVE ea,ea
LET D1=D0                      MOVE D0,D1
LET SP=SP-4:POKE(SP),D7        MOVE D7,-(SP)
POKE(SP),D7:LET SP=SP+4        MOVE (SP)+,D7
```

Use MOVEA where the destination is an address register.

Instruction summary – continued

MOVE from **SR:** Save the word contents of the status register in the effective address register or memory location.
Be careful as this instruction is privileged in the MC68010 and MC68020 instruction sets, programmers should try not to use it in user state.

```
                                MOVE SR,ea
LET D0=PEEK_W(SR)               MOVE.W SR,D0
```

MOVE to **CCR:** Move the contents of the source operand WORD into the condition code register.
Only the low-order byte is used; the upper byte is ignored.

```
                                MOVE ea,CCR
POKE_W(CCR),4                   MOVE #4,CCR
```

Set the Zero flag and clear all others.

MOVE to **SR:** Move the contents of the source operand into the status register.
This is a privileged instruction and attempted access while in user mode will cause a trap to the privilege violation exception vector.

```
                                MOVE ea,SR
POKE_W(SR),1792                 MOVE #1792,SR
```

Clear all flags, set user state, and set interrupt mask to level seven.

MOVE USP: Move the contents of the user stack pointer to or from the specified address register.
This is a privileged instruction and attempted access while in user mode will cause a trap to the privilege violation exception vector.

```
LET A3=USP                      MOVE USP,A3
LET USP=A3                      MOVE A3,USP
```

MOVEA: Move the contents of the source effective address to the destination address register.
Byte-sized operations are not permitted.

```
LET A3=PEEK_W(192)              MOVEA.W 192,A3
LET A0=PEEK_L(4)                MOVEA.L  4,A0
```


Instruction summary – continued

MOVEM: Move multiple registers to or from memory which permits the transfer of a block of specified registers to and from memory in a predetermined sequence by one instruction.

```
LET A7=A7-4:POKE_L(A7),D0                MOVEM.L #57344,-(A7)
LET A7=A7-4:POKE_L(A7),D1
LET A7=A7-4:POKE_L(A7),D2
                                         MOVEM.L (A7)+,#1860
```

Saves registers D0, D1 and D2 in order on the stack and then recovers them in the reverse order.

These instructions save registers D0, D1 and D2

```
                                         (MOVEM.L # 7,24(A7)
                                         (MOVEM.L #57344, -(A7) *
```

These two instructions recover registers D0, D1 and D2

```
                                         (MOVEM.L 24(A7),#7
                                         (MOVEM.L (A7)+,#7
```

* The predecrement mode of addressing values the registers in reverse order for the register list mask, permitting push-on, pull-off on a last in-first out basis.

MOVEP: Move data to or from a data register and alternate bytes in memory.
Enables the MC68000 to interface with 8-bit peripheral devices. The data is transferred on either the high half of the data bus D8-D15, even addresses, or the low half D0-D7, odd addresses, to memory occupying alternate bytes in the processor's memory map. The data is transferred in high-low order.

```
POKE_W(7+65536),Dn                       MOVEP Dn,d(Ay)
LET Dn=PEEK_W(7+65536)                   MOVEP 7(Ay),Dn
```

This is the *only* instruction that provides word and longword access at odd addresses.

MOVEQ: Move sign-extended 32-bit immediate data in the range of +127 to –128 to a data register.
A fast means of loading small positive and negative integers into a data register.

```
LET D0=0                                 MOVEQ #0,D0
```

Instruction summary – continued

MULS: Multiply two signed 16-bit operands.
Only the low-order 16-bits are used from both operands for the multiplication, the result being the 32-bit product in the destination data register.

```
MULS ea,Dn
```

ASL ea is a fast signed multiply by two.

MULU: Multiply two unsigned 16-bit operands.
Only the low-order 16-bits are used from both operands for the multiplication, the result being the 32-bit product in the destination data register.

```
MULU ea,Dn
```

NBCD: Negate Decimal with Extend.
Subtracts the destination byte-sized operand and the extend bit from zero using decimal arithmetic.

```
NBCD ea
```

Extend bit clear, the ten's complement is produced.
Extend bit set, the one's complement is produced.

NEG: Negate subtracts the destination operand from zero, producing the two's complement of a byte, word or longword operand.

```
NEG ea
```

NEGX: Negate with extend subtracts the destination operand and the extend bit from zero, producing the two's complement of a byte, word or longword operand.

```
NEGX ea
```

NOP: No OPeration.
Has no effect other than to increment the program counter by 2. Its use is generally either for creating a space in code which may be used later on for adding a subroutine call, for writing text etc. or for deleting parts of code, especially test routines, without the need for recompiling.

```
NOP
```

Instruction summary – continued

NOT: Logically complement.
Produces the one's complement of the operand.

```
NOT ea
```

OR: Or the source to the contents of the destination data register.
The source OR data is normally used to set specific bits of an operand.

```
LET Dn=src || Dn                    OR ea,Dn
```

If src = 3, then OR src sets bits 0 and 1 in Dn; the other bits are left unchanged.

Use OR ea,Dn where the destination is a data register.
Use ORI where the source is immediate data.

ORI: ORI the immediate data to the destination effective address.

```
                                    ORI.W #512,D0
LET ea=data || ea          Set bit 9 of word, others unchanged
```

ORI to CCR: ORI the data to the condition code register.

```
LET CCR=5 || CCR                    ORI #5,CCR
```

ORI is used to set bit positions: Zero and Carry (bits 0 and 2 in the CCR) are set, the others are unchanged.

ORI to SR: ORI the data to the status register

```
LET SR=1792 OR SR                   ORI #1792,SR
```

Set the status register interrupt mask to level seven, all other conditions unchanged. This is a privileged instruction and attempted access from user mode will cause a trap to the privilege violation exception process routine.

Instruction summary – continued

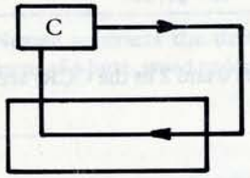
PEA: Push effective address.
Pushes a longword-computed address onto the current stack. It is useful for passing parameters to a subroutine which are accessed via an address register indirect with displacement instruction and removed from the stack prior to return if necessary.

```
                                PEA param
                                JSR sprog

Access parameter        sprog   MOVEA.L 4(SP),A0
Tidy stack                      MOVE.L (SP)+,(SP)

                                RTS
```

RESET: Reset external devices by asserting the reset line.
There is no affect on the processor other than an increase of two in the value of the program counter. This is a privileged instruction and attempted access while in user mode will cause a trap to the privilege violation exception vector.

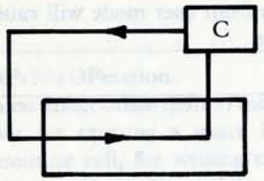
```
RESET
```

ROL: ROtate without extend Left.
The MSB is rotated to the LSB and the carry; the other bits are shifted up one.




```
ROL ea (shift 1)
ROL Dx,Dy (Reg modulo 64)
ROL #5,Dy (shift 1 to 8)
```

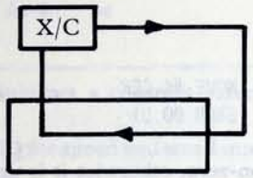
ROR: ROtate without extend Right.
The LSB is rotated to the MSB and the carry; the other bits are shifted down one.




```
ROR ea (shift 1)
ROR Dx,Dy (reg modulo 64)
ROR #5,Dy (shift 1 to 8)
```

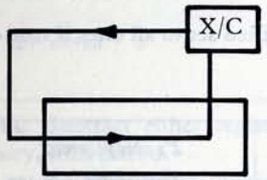

Instruction summary – continued

ROXL: ROtate with eXtend Left.
The MSB is rotated to the extend bit and the carry, the extend bit is rotated to the LSB and the other bits are shifted up one.




```
ROXL ea (shift 1)
ROXL Dx,Dy (reg modulo 64)
ROXL #5,Dy (shift 1 to 8)
```

ROXR: ROtate with eXtend Right.
The LSB is rotated to the extend bit and the carry, the extend bit is rotated to the MSB and the other bits are shifted down one.




```
ROXR ea (shift 1)
ROXR Dx,Dy (reg modulo 64)
ROXR #5,Dy (shift 1 to 8)
```

RTE: Return from Exception.
The status register and the program counter are pulled from the current (supervisor) stack. This instruction is privileged and attempted access while in user mode will cause a trap to the privilege violation exception vector.

```
(SP)+,SR    RTE
(SP)+,PC
```

RTR: Return and Restore.
The condition code and then the program counter are pulled from the current stack.

```
(SP)+,CCR   RTR
(SP)+,PC
```

RTS: Return from Subroutine. The program counter is pulled from the current stack.

```
(SP)+,PC    RTS
```

Instruction summary – continued

SBCD: Subtract Decimal with Extend.
Subtract a byte-sized binary coded decimal number and the extend bit from the destination operand byte using decimal arithmetic and store the result in the destination location.

```
SBCD $7        SBCD $27
     $6             $16          MOVE #4,CCR
     $1             $11          SBCD D0,D1
```

Note that the z-flag is cleared if the result is non-zero, otherwise it is unchanged. For memory additions the data must be stored with the most significant digit lower in memory and the address register pointers initially set to the byte above the low-order BCD digit. The only memory addressing mode is predecrement.

Scc: Set according to condition.
The specified condition is tested and the byte specified set to all ones if true or all zeros if false. The condition may be:

		2's complement arithmetic
EQ Equal To	CS Carry Set	GT Greater than
NE Not Equal	CC Carry Clear	LT Less than
MI Minus	VS Overflow	GE Greater than or Equal to
PL Plus	VC No Overflow	
HI Higher than	T True	LE Less than or Equal to
LS Lower than or same	F False	

Scc ea

STOP: Load the status register and Stop.
The immediate operand is put into the status register and the program counter advanced to the next instruction and then stopped. Execution only resumes when a trace, interrupt or reset exception occurs.

```
STOP #7
```

SUB: Subtract the source from the destination.
One of the integers must be the contents of a data register.

```
LET Dn=Dn - ea        SUB ea,Dn
LET ea=ea - Dn        SUB Dn,ea
```

Use SUB ea,Dn where the destination is a data register.
Use SUBA where the destination is an address register.
Use SUBI or SUBQ where the source is immediate data.

Instruction summary – continued

SUBA: Subtract the contents of the effective address from the contents of the destination address register.

```
LET An=An - ea        SUBA ea,An
```

SUBI: Subtract a constant value from the contents of the destination effective address.
Use SUBQ for speed and small integers.

```
LET ea=ea - 999       SUBI #999,ea
```

SUBQ: Subtract a constant of from 1 to 8 from the contents of the effective address. Faster subtraction than SUBI.

```
LET ea=ea - 8         SUBQ #8,ea
```

SUBX: Subtract either register to register, or predecremented memory from memory, with extend.
The extend bit enables multiprecision arithmetic to be performed, acting as a borrow between successive operations.

Memory subtractions
SUBX -(Ay),-(Ax)
Where X infers the Extend bit
LET Ay=Ay-4
LET Ax=Ax-4

Data register subtractions
Subtract two 64 bit integers
D0_D1 and D2_D3 Lo-Hi resply
SUB.L D0,D2 Low bits
SUBX.L D1,D3 High bits

POKE(Ax),PEEK(Ax) - PEEK(Ay) - X

Memory addition

```
MOVE   #4,CCR
SUBX.L -(A0),-(A1)
SUBX.L -(A0),-(A1)
```

Note that the z-flag is cleared if the result is non-zero, otherwise it is unchanged. For memory additions first clear the Extend bit and set the Zero flag. The data must be stored with the most significant digit lower in memory and the address pointers initially set the operand size above the low order digit. Predecrement is the only memory addressing mode.

SWAP: Swap register halves exchanges the high-order word of a data register with the low-order word.
This instruction provides access to the low-order byte of the high word.

```
Dn 0--15 < - - > Dn 16-31
```

Instruction summary – continued

TAS: Test and set an operand, compares the operand byte with zero and sets the condition codes accordingly.
If the byte is zero, the Z_flag is set; if the MSB is non-zero, then the N_flag is set. The MSB of the operand is then set.

```
TAS ea
```

TRAP: Trap. The processor commences execution at the relevant trap exception vector address.

```
TRAP #n
```

TRAPV: Trap on Overflow.
The processor commences execution at the trap on overflow exception vector address.

```
TRAPV
```

TST: Test an operand.
The operand is compared with zero and the condition codes set accordingly.

```
TST ea
```

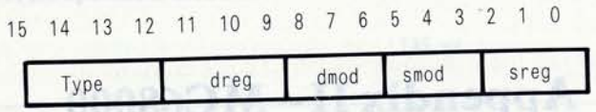
Use in preference to CMPI #0,ea

UNLK: Unlink.
The stack pointer is loaded from the specified address register; the address register is then loaded with the longword pulled from the top of the stack and the linked space deallocated.

```
UNLK An
```

Appendix H – MC68000 Instruction Codes

The Motorola MC68000 series of microprocessors rationalize instruction code allocation by segmenting the 16-bit Operation Word into five smaller blocks, each of which has a fairly consistent meaning. Operation word instruction



Instruction Word Parsing Analysis

Type

The types 0 to 15 instruction codes (16 classes) are allocated as follows:

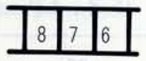
Type	*Instructions Range*
0	Bit manipulation, Move Peripheral and Immediate instructions
1	Move byte Instructions
2	Move longword instructions
3	Move word instructions
4	Miscellaneous instructions
5	Add Quick, Subtract Quick, Set conditionally and Decrement instructions
6	Branch conditionally instructions
7	Move Quick instructions
8	Or, Divide and Subtract decimal instrs.
9	Subtract, Subtract extended instrs.
10	Unassigned
11	Compare, Exclusive Or instructions
12	And, Multiply, Add decimal and Exchange instructions
13	Add, Add extended instructions
14	Shift and Rotate instructions
15	Unassigned

Dreg

Dreg has three main uses, normally holding the destination address in the general move instruction, one of the two register numbers for use in the specified instruction or embedded data for use in the add and subtract quick instructions.

Dreg only refers to a register in those instances where the instruction has two register operands.

Dmod



Dmod has two main uses, specifying the effective address mode of the destination operand in the general move instruction or, in most other cases it defines the size of the operation to be performed.

Smod

Smod usually defines the effective address mode of the instruction, the source operand for the move instruction.

Sreg

Sreg defines the effective address register, usually the source.

Instruction codes

BIT MANIPULATION, MOVE PERIPHERAL AND IMMEDIATE INSTRUCTIONS

Type 1 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
BCHG Dn,ea	Dn	5	– ea –		dataltadd	- - A - -
BCHG data,ea	4	1	– ea –		dataltadd	- - A - -
BCLR Dn,ea	Dn	6	– ea –		dataltadd	- - A - -
BCLR data,ea	4	2	– ea –		dataltadd	- - A - -
BSET Dn,ea	Dn	7	– ea –		dataltadd	- - A - -
BSET data,ea	4	3	– ea –		dataltadd	- - A - -
BTST Dn,ea	Dn	4	– ea –		dataddmd2	- - A - -
BTST data,ea	4	0	– ea –		dataddmd2	- - A - -
MOVEP Dx,d(Ay)	Dx	1 1 x	1	Ay	-	- - - - -
MOVEP d(Ay),Dx	Dx	1 0 x	1	Ay	-	- - - - -
ORI data,ea	0	0 s s	– ea –		dataltadd	- A A 0 0
ORI data,CCR	0	0	7	4	-	A A A A A
ORI data,SR	0	1	7	4	-	A A A A A
ANDI data,ea	1	0 s s	– ea –		dataltadd	- A A 0 0
ANDI data,CCR	1	0	7	4	-	A A A A A
ANDI data,SR	1	1	7	4	-	A A A A A
SUBI data,ea	2	0 s s	– ea –		dataltadd	A A A A A
ADDI data,ea	3	0 s s	– ea –		dataltadd	A A A A A
EORI data,ea	5	0 s s	– ea –		dataltadd	- A A 0 0
EORI data,CCR	5	0	7	4	-	A A A A A
EORI data,SR	5	1	7	4	-	A A A A A
CMPI data,ea	6	0 s s	– ea –		dataltadd	- A A A A

MOVE BYTE INSTRUCTION

Type 1 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
MOVE.B ea,ea						– A A 0 0
source		– ea –			ALL *	
destination	– ea –				dataltadd	

* Address register direct mode is not permitted

MOVE LONGWORD INSTRUCTION

Type 2 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
MOVE.L ea,ea						– A A 0 0
source		– ea –			ALL	
destination	– ea –				dataltadd	

MOVE WORD INSTRUCTION

Type 3 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
MOVE.W ea,ea						– A A 0 0
source		– ea –			ALL	
destination	– ea –				ataltadd	

MISCELLANEOUS INSTRUCTIONS

Type 4 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
NEGX ea	0	0 s s	– ea –		dataltadd	A A A A A
CLR ea	1	0 s s	– ea –		dataltadd	– 0 1 0 0
NEG ea	2	0 s s	– ea –		dataltadd	A A A A A
NOT ea	3	0 s s	– ea –		dataltadd	– A A 0 0
MOVE SR,ea	0	3	– ea –		dataltadd	– – – – –
MOVE ea,CCR	2	3	– ea –		dataddmd1	A A A A A
MOVE ea,SR	3	3	– ea –		dataddmd1	A A A A A
SWAP Dn	4	1	0	Dn	—	– A A 0 0
EXT.W Dn	4	2	0	Dn	—	– A A 0 0
EXT.L Dn	4	3	0	Dn	—	– A A 0 0
NBCD.B ea	4	0	– ea –		dataltadd	A u A u A
PEA ea	4	1	– ea –		conaddmd1	– – – – –
MOVEM list,ea	4	0 1 x	– ea –		conaltadd	– – – – –
MOVEM ea,list	6	0 1 x	– ea –		conaddmd2	– – – – –
TST ea	5	0 s s	– ea –		dataltadd	– A A 0 0
TAS ea	5	3	– ea –		dataltadd	– A A 0 0
ILLEGAL	5	3	7	4	—	– – – – –
TRAP data	7	1	0 0 v v v v		—	– – – – –
LINK An,data	7	1	2	An	—	– – – – –
UNLK An	7	1	3	An	—	– – – – –
MOVE An,USP	7	1	4	An	—	– – – – –
MOVE USP,An	7	1	5	An	—	– – – – –
RESET	7	1	6	0	—	– – – – –
NOP	7	1	6	1	—	– – – – –
STOP data	7	1	6	2	—	A A A A A
RTE	7	1	6	3	—	A A A A A
RTS	7	1	6	5	—	– – – – –
TRAPV	7	1	6	6	—	– – – – –
RTR	7	1	6	7	—	A A A A A

Type 4 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*

use on on H5 & H7

x Size	*s s Size*	*Condition codes*
0 = Word	0 0= Byte	u = Undefined
1 = Longword	0 1= Word	A = Affected
	1 0= Longword	– = Unaffected
ea = Effective address		0 = Cleared
CCR = Condition code register		1 = Set
SR = Status register		

ADD AND SUBTRACT QUICK, SET CONDITIONALLY and DECREMENT INSTRUCTIONS

Type 5 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
AddQ data,ea	data	0 s s	– ea –		altaddmod	A A A A A
SUBQ data,ea	data	1 s s	– ea –		altaddmod	A A A A A
Scc ea	c c c c	1 1	– ea –		dataltadd	– – – – –
DBcc Dn,data	c c c c	1 1	1	Dn	—	– – – – –

x Size	*s s Size*	*Condition codes*
0 = Word	0 0= Byte	u = Undefined
1 = Longword	0 1= Word	A = Affected
	1 0= Longword	– = Unaffected
ea = Effective address		0 = Cleared
CCR = Condition code register		1 = Set
SR = Status register		
c c c c = 4–bit Condition code		
v v v v = 4–bit Vector address		

BRANCH CONDITIONALLY INSTRUCTION

Type 6 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
Bcc data	c c c c	displacement			—	– – – – –
BSR data	0	1	displacement		—	– – – – –
BRA data	0	0	displacement		—	– – – – –

x Size	*s s Size*	*Condition codes*
0 = Word	0 0= Byte	u = Undefined
1 = Longword	0 1= Word	A = Affected
	1 0= Longword	– = Unaffected
ea = Effective address		0 = Cleared
CCR = Condition code register		1 = Set
SR = Status register		

CONDITIONAL TESTS

cc	*Mnemonic*	*Condition*
0	T	TRUE
1	F	FALSE
2	HI	HIGH
3	LS	LOW or SAME
4	CC	CARRY CLEAR
5	CS	CARRY SET
6	NE	NOT EQUAL
7	EQ	EQUAL
8	VC	OVERFLOW CLEAR
9	VS	OVERFLOW SET
10	PL	PLUS
11	MI	MINUS
12	GE	GREATER or EQUAL
13	LT	LESS THAN
14	GT	GREATER THAN
15	LE	LESS or EQUAL

There is no Branch TRUE BT or Branch FALSE BF, the codes are used by the BSR and BRA instructions

MOVE QUICK INSTRUCTIONS

Type 7 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
MOVEQ data,Dn	Dn	0	data	—	–	A A 0 0

2's complement data value

OR, DIVIDE AND SUBTRACT DECIMAL INSTRUCTIONS

Type 8 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
OR ea,Dn	Dn	0 s s	– ea –		dataddmdl	– A A 0 0
OR Dn,ea	Dn	1 s s	– ea –		altmemadd	– A A 0 0
DIVU ea,Dn	Dn	3	– ea –		dataddmdl	– A A A 0
DIVS ea,Dn	Dn	7	– ea –		dataddmdl	– A A A 0
SBCD Dy,Dx	Dx	4	0	Dy	—	A u A u A
SBCD –(Ay), –(Ax)	Ax	4	1	Ay	—	A u A u A

x Size	*s s Size*	*Condition codes*
0 = Word	0 0= Byte	u = Undefined
1 = Longword	0 1= Word	A = Affected
	1 0= Longword	– = Unaffected
ea = Effective address		0 = Cleared
CCR = Condition code register		1 = Set
SR = Status register		

SUBTRACT AND SUBTRACT EXTENDED INSTRUCTIONS

Type 9 (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
SUBA.W ea,An	An	3	– ea –		ALL	– – – – –
SUBA.L ea,An	An	7	– ea –		ALL	– – – – –
SUB ea,Dn	Dn	0 s s	– ea –		ALL	A A A A A
SUB Dn,ea	Dn	1 s s	– ea –		altmemadd	A A A A A
SUBX Dy,Dx	Dx	1 s s	0	Dy	—	A A A A A
SUBX –(Ay),–(Ax)	Ax	1 s s	1	Ay	—	A A A A A

EMULATION INSTRUCTIONS – Type 10 (#$A)

Normally available for the implementation of user–written routines and entered by ensuring four MSB of the op word or defined word constant are 1010 (10 dec), which will cause a trap to a user routine; other bits of op word may be used for parameter passing. The ST uses this instruction for initializing and operating the 'A'–line functions on which GEM VDI and subsequently GEM AES are based – so use with care.

COMPARE AND EXCLUSIVE OR INSTRUCTIONS

Type 11 (#$B) (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
CMPA ea,An	An	x 1 1	– ea –		ALL	– A A A A
CMP ea,Dn	Dn	0 s s	– ea –		ALL	– A A A A
CMPM –(Ay),–(Ax)	Ax	1 s s	1	Ay	—	– A A A A
EOR Dn,ea	Dn	1 s s	– ea –		dataltadd	– A A 0 0

x Size
0 = Word
1 = Longword

s s Size
0 0= Byte
0 1= Word
1 0= Longword

Condition codes
u = Undefined
A = Affected
– = Unaffected
0 = Cleared
1 = Set

ea = Effective address
CCR = Condition code register
SR = Status register

AND, MULTIPLY, ADD DECIMAL AND EXCHANGE INSTRUCTIONS

Type 12 (#$C) (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
AND ea,Dn	Dn	0 s s	– ea –		datadmd1	– A A 0 0
AND Dn,ea	Dn	1 s s	– ea –		altmemadd	– A A 0 0
MULU ea,Dn	Dn	3	– ea –		datadmd1	– A A 0 0
MULS ea,Dn	Dn	7	– ea –		datadmd1	– A A 0 0
ABCD Dy,Dx	Dx	4	0	Dy	—	A u A u A
ABCD –(Ay),–(Ax)	Ax	4	1	Ay	—	A u A u A
EXGD Dx,Dy	Dx	5	0	Dy	—	– – – – –
EXGA Ax,Ay	Ax	5	1	Ay	—	– – – – –
EXGM Dx,Ay	Dx	6	1	Ay	—	– – – – –

ADD AND ADD EXTENDED INSTRUCTIONS

Type 13 (#$D) (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
ADDA.W ea,An	An	3	– ea –		ALL	– – – – –
ADDA.L ea,An	An	7	– ea –		ALL	– – – – –
ADD ea,Dn	Dn	0 s s	– ea –		ALL	A A A A A
ADD Dn,ea	Dn	1 s s	– ea –		altmemadd	A A A A A
ADDX Dy,Dx	Dx	1 s s	0	Dy	—	A A A A A
ADDX –(Ay),–(Ax)	Ax	1 s s	1	Ay	—	A A A A A

x Size
0 = Word
1 = Longword

s s Size
0 0= Byte
0 1= Word
1 0= Longword

Condition codes
u = Undefined
A = Affected
– = Unaffected
0 = Cleared
1 = Set

ea = Effective address
CCR = Condition code register
SR = Status register

SHIFT AND ROTATE INSTRUCTIONS

Type 14 (#$E) (Bits 15–12)

Instruction syntax	*Dreg 11–9*	*Dmod 8–6*	*Smod 5–3*	*Sreg 2–0*	*Address Mode*	*Cond Codes X N V C*
ASL Dx,Dy	Dx	1 s s	4	Dy	—	A A A A A
ASL data,Dy	count	1 s s	0	Dy	—	A A A A A
ASL ea	0	7	– ea –		altmemadd	A A A A A
ASR Dx,Dy	Dx	0 s s	4	Dy	—	A A A A A
ASR data,Dy	count	0 s s	0	Dy	—	A A A A A
ASR ea	0	3	– ea –		altmemadd	A A A A A
LSL Dx,Dy	Dx	1 s s	5	Dy	—	A A A 0 A
LSL data,Dy	count	1 s s	1	Dy	—	A A A 0 A
LSL ea	1	7	– ea –		altmemadd	A A A 0 A
LSR Dx,Dy	Dx	0 s s	5	Dy	—	A A A 0 A
LSR data,Dy	count	0 s s	1	Dy	—	A A A 0 A
LSR ea	1	3	– ea –		altmemadd	A A A 0 A
ROL Dx,Dy	Dx	1 s s	7	Dy	—	– A A 0 A
ROL data,Dy	count	1 s s	3	Dy	—	– A A 0 A
ROL ea	3	7	– ea –		altmemadd	– A A 0 A
ROR Dx,Dy	Dx	0 s s	7	Dy	—	– A A 0 A
ROR data,Dy	count	0 s s	3	Dy	—	– A A 0 A
ROR ea	3	3	– ea –		altmemadd	– A A 0 A
ROXL Dx,Dy	Dx	1 s s	6	Dy	—	A A A 0 A
ROXL data,Dy	count	1 s s	2	Dy	—	A A A 0 A
ROXL ea	2	7	– ea –		altmemadd	A A A 0 A
ROXR Dx,Dy	Dx	1 s s	6	Dy	—	A A A 0 A
ROXR data,Dy	count	1 s s	2	Dy	—	A A A 0 A
ROXR ea	2	3	– ea –		altmemadd	A A A 0 A

x Size

0 = Word
1 = Longword

s s Size

0 0= Byte
0 1= Word
1 0= Longword

Condition codes

u = Undefined
A = Affected
– = Unaffected
0 = Cleared
1 = Set

ea = Effective address
CCR = Condition code register
SR = Status register

EMULATION INSTRUCTION – Type 15 (#$F)

Normally available for the implementation of user–written routines, and entered by ensuring four MSB of the op word or defined word constant are 1111 (15 dec), directing the trap service to a user routine. Other bits of op word may be used for parameter passing.

This service trap is used by the MC68020 processor for passing co–processor instructions. The ST uses it in processing the application environment services (AES), so be careful.

Appendix I – Error codes

BIOS ERROR CODES

Error code	*Function*	*Comments*
0	OK	Successful operation
–1	Error	
–2	Drive not ready	Not ready, not attached or busy
–3	Unknown command	Command not understood by device
–4	CRC error	Soft error while reading sector
–5	Bad request	Bad parameter, Cannot do request
–6	Seek error	Drive could not seek
–7	Unknown media	Foriegn media. Bad zero boot sect
–8	Sector not found	
–9	No paper	
–10	Write fault	
–11	Read fault	
–12	General error	Reserved
–13	Write protect	Read only or protected media
–14	Media change	Media changed since last write or the rd/wr op not done (file error)
–15	Unknown device	BIOS doesn't recognize device
–16	Bad sectors	Format yielded bad sectors
–17	Insert disk	Disk not in drive (shell error)

BDOS ERROR CODES

Error code	*PC DOS equiv*	*Function*	
		Supported	*Not supported*
–32	1	Invalid function number	
–33	2	File not found	
–34	3	Path not found	
–35	4	No handles left (too many open files)	
–36	5	Access denied	
–37	6		Invalid handle
–38	7		
–39	8	Insufficient memory	
–40	9		Invalid memory block address
–41	10	** Insufficient memory	
–42	11	** Insufficient memory	
–43	12		
–44	13		
–45	14		
–46	15	Invalid drive specified	
–47	16	** Invalid operation	
–48	17		
–49	18	No more files	

The list of PC-DOS equivalent error codes supported may be found by running the GEM demonstration program (Appendix L).

MISCELLANEOUS ERROR CODES

Error code	*Function*
–64	Range error
–65	Internal error
–66	Invalid program load format
–67	Setblock failure due to growth restrictions

Appendix J – BASIC GEM

ST BASIC provides the programmer with direct access to parts of the operating system AES and VDI interface.

GEMSYS

The AES control arrays are accessed through the AES parameter block (GB pointer), the block provides pointers to the other supplementary AES parameter blocks:

Control table	+$0
Global array	+$4
Int_in table	+$8
Int_out table	+$C
Addr_in table	+$10
Addr_out table	+$14

Data input and output as specified in the AES traps and utility tables Chapter 3.

The tables are used by the programmer to input data, call the appropriate GEM AES function, GEMSYS(n), and read any reply from the data placed in the output tables by the function.

VDISYS

The VDI parameter blocks are directly accessible from BASIC:

contrl	input	tables
ptsin	input	
ptsout	output	
intin	input	
intout	output	

The appropriate tables are loaded with data and the function called via VDISYS(1), the (1) being a dummy argument. Any reply is read from the output tables.

GEMSYS			VDISYS
GB	control	Indirect access	contrl
	global		ptsin
	int_in		ptsout
	int_out		intin
	addr_in		intout
	addr_out		

SYSTAB

ST BASIC also provides access to a BASIC system table of the following read only pointers and parameters:

Graphics resolution	+$0	1 = high resolution 2 = medium resolution 4 = low resolution
Editor ghost line style (Read/write)	+$2	0 = thickened 1 = intensity 2 = skewed 3 = underlined 4 = outline 5 = shadow
Edit AES handle	+$4	1 (default)
List AES handle	+$6	2 (default)
Output AES handle	+$8	3 (default)
Command AES handle	+$A	4 (default)
Edit open flag	+$C	0 = closed, 1 = open
List open flag	+$E	0 = closed, 1 = open
Output open flag	+$10	0 = closed, 1 = open
Command open flag	+$12	0 = closed, 1 = open
Graphics buffer	+$14	Longword 32K buffer pntr
GEM flag	+$18	0 = normal, 1 = off

The GEM flag is used to turn BASIC I/O off and increase the processing speed of GEM based operations. With BASIC partially off, the I/O functions involving the screen, mouse and keyboard are disabled, although disk I/O is still enabled.

The BASIC functions can be re-enabled after the burst of speed for user input

Not all GEM and VDI functions are available through BASIC, some of the BASIC housekeeping activities negate the effect of the functions.

Cautionary notes:

Ensure that evaluations of the graphic primitives take into account color. Many experiments may appear not to work simply because the writing color is the same as the screen backgound.

Characters are written to the screen starting from the left-hand edge and will probably be obscured by the command screen border unless the programmer moves it out of the way.

Use the mouse and the right button to draw a primitive, use the left button to change the primitive. Note the effect on a primitive of crossing the left hand screen edge.

```
10   start: CLEAR: a#=gb:int_out=PEEK(a#+12)
20   FULLW 2:CLEARW 2
30   INPUT "GDP (1 to 9) ";gdp
40   IF gdp<1 or gdp>9 THEN GOTO start
50   POKE systab+24,1 : REM BASIC I/O off
60   POKE contrl,122:POKE contrl+2,0:POKE contrl+6,1
70   GOSUB curson
80   attribs: GEMSYS(79)
90   x=PEEK(int_out+2)    : REM x mouse
100  y=PEEK(int_out+4)    : REM y mouse
110  key=PEEK(gintout+6) : REM button state nil_left_right
120  ON key+1 GOSUB showcurs, done, drawprim
130  GOTO attribs
140  done: POKE systab+24,0:GOTO start:REM nasty return
150  drawprim: REM
160  COLOR 1,(RND*15)+1,1,RND*25,2 : REM random color
170  IF mouse=0 THEN GOTO 210
180  mouse=0
190  POKE contrl,123:POKE contrl+2,0:POKE contrl+6,0
200  VDISYS(1) : REM hide cursor
210  POKE contrl,11:in=0:xop=x+50:yop=y+50:rc=0
220  ptin=2:IF gdp=4 THEN ptin=3:xop=0:yop=0:rc=50
230  IF gdp=2 OR gdp=3 THEN ptin=4:xop=0:yop=0:in=2
240  POKE contrl+2,ptin
250  IF gdp=6 OR gdp=7 THEN xop=50:yop=20:in=2
260  IF gdp=5 THEN xop=60:yop=40
270  IF in<>0 THEN POKE contrl+6,in
280  POKE contrl+10,gdp
290  POKE ptsin,x
300  POKE ptsin+2,y
310  POKE ptsin+4,xop
320  POKE ptsin+6,yop
330  REM IF ptin=2 THEN GOTO nxtin
340  POKE ptsin+8,rc
350  POKE ptsin+10,0
360  REM IF ptin=3 THEN GOTO nxtin
370  POKE ptsin+12,50
380  POKE ptsin+14,0
390  REM nxtin: IF in=0 THEN GOTO draw
400  POKE intin,(rnd*3600)
410  POKE intin+2,(rnd*3600)
420  draw: VDISYS(1)
430  RETURN
440  showcurs: IF mouse=1 THEN RETURN
450  POKE contrl,122:POKE contrl+2,0:POKE contrl+6,0
460  curson: POKE intin,0: VDISYS(1)
470  mouse=1: RETURN
```

Look at the spelling of the variables, particularly contrl, if the program crashes. Although BASIC access to the processor is normally in user mode, PEEK and POKE instructions are performed in supervisor mode to provide access to all parts of memory.

BASIC ASSEMBLER

There are many ways of producing a combined BASIC/assembler program on the Atari ST computer, the following demonstrates one of them:

First create the assembler subroutine of relocatable 68000 machine code that can be saved using a BASIC program similar to the following.

```
10  RESTORE
20  ZA$="12345678901234567890": REM \Use either method to
30  ZB$=STRING$(100,"*"):       REM /create space for code
40  y=VARPTR(ZA$):              REM Somewhere to put code
50  DEF SEG=y:                  REM Set up loop offset
60  FOR a=0 TO n
70  READ x:POKE a,x:            REM Put code into memory
80  NEXT a
90  BSAVE "prog1.asm",y,n:      REM Save code to disk
100 STOP
200 DATA . . . . .              REM Byte sized data
```

The machine code will probably be loaded into a space created within the main BASIC program by a dummy variable. Obviously, any number of different machine code utilities can be loaded into the same space dependant upon program state, or they may be stored in individual program spaces.

Parameters are passed to the machine code routine on the user stack which contains an integer count of the number of parameters passed on top. The next item on the stack is a longword pointer to the 8-byte per parameter array. String variables use the array parameter as a pointer to the string.

Output can be placed in predefined variables and if correctly formatted, read back by the BASIC program.

```
10  ZB$=STRING$(100,"*"):       REM Space to load code
20  ZR$="12345678":             REM Space for reply
30  y=VARPTR(ZB$):              REM Position for code
40  ans=VARPTR(ZR$):            REM Position of reply
50  BLOAD "prog1.asm",y:        REM Load code from disk
.
.
100 CALL prog1(x,y,ans):        REM Call program code,
                                passing parameters x
                                and y, returning data
                                in the variable ans
```

An alternative might be to compile the code within the BASIC program proper if the machine code program length is quite short.

HAND CODING

Many programmers had their first contact with assembly language programming through hand coded 8-bit microprocessor routines embedded in short BASIC programs. MC68000 code is slightly more complicated to assemble than 8-bit code, but is still perfectly manageable.

Use tables of instruction types 0 to 15 (H5 to H14), to generate the basic code i.e:

$$4096 \times \texttt{type} + 512 \times \texttt{dreg} + 64 \times \texttt{dmod} + 8 \times \texttt{smod} + \texttt{sreg}$$

and the address mode encoding table (H.15) to determine the effective address (-ea-) values if required.

Example of a hand coded program

Project: Monitor screen inversion Author:
Version: 2 Date: Dec /85

labl	Syntax	Src Mnm	Dest Mnm	type	dmod / dreg	sreg / smod			dec value	Notes
00	MOVE.W	n	–(SP)	3	7	4	7	4	16188	Get old color
02			–1						–1	
04	MOVE.W	n	–(SP)	3	7	4	7	4	16188	
06			0						0	
08	MOVE.W	n	–(SP)	3	7	4	7	4	16188	
10			7						7	
12	TRAP	14		4	7	1		14	20046	
14	ADDA.W	n	SP	13	7	3	7	4	57084	Tidy stack
16			6						6	
18	EORI.W	n	D0	0	5	1	0	0	2624	Toggle col bit 0
20			1						1	
22	MOVE.W	D0	–(SP)	3	7	4	0	0	16128	Set new colour
24	MOVE.W	n	–(SP)	3	7	4	7	4	16188	
26			0						0	
28	MOVE.W	n	–(SP)	3	7	4	7	4	16188	
30			7						7	
32	TRAP	14		4	7	1		14	20046	
34	ADDA.W	n	SP	13	7	3	7	4	57084	Tidy stack
36			6						6	
38	RTS			4	7	1	6	5	20085	Return to BASIC

Use this type of program to load the code into a file on disk:

```
10   restore:n=70
20   zb$=string$(100,"*")
30   y=varptr(zb$)
40   def seg = y
50   for a=0 to n step 2
60   read x:poke a,int(x/256):poke a+1,x mod 256
70   next a
80   bsave "b/w.asm",y,n+2
100  stop
210  data 16188,-1,16188,0,16188,7
220  data 20046,57084,6,2624,1
230  data 16128,16188,0,16188,7
240  data 20046,57084,6
250  data 20085
300  data 0,0,0,0,0,0,0,0,0,0,0,0
310  data 0,0,0,0,0,0,0,0,0,0,0,0
320  data 0,0,0,0,0,0,0,0,0,0,0,0
```

and to toggle the screen or border color, run the following BASIC program which loads the file back from disk and executes it:

```
10   zb$=string$(100,"*")
20   y=varptr(zb$)
30   bload "xb/w.asm",y
40   call y
50   stop
```

The following brief notes may be useful in compiling programs in the above manner:

Entry to machine code level from BASIC is in supervisor mode.

If you drop to user mode, be careful where you place your stack. Perhaps you might like to use the following sequence of instructions that jump over your stack or data to the beginning of the executable code. DATA 17402,4,24576 + dis,...

Start	LEA 4(PC),A1	Set A1 to start of text
	BRA dis	dis = 2 + text length (must be even)
	Text or stack	
	Program code	Start of program

If in difficulties with a BRAnch or JuMP, surround with NOP's to make the jump less sensitive to the count.

Project: Author: Date:
Version:

	labl	Syntax	Src Mnm	Dest Mnm	type	dreg	dmod	smod	sreg	dec value	Notes
0											
2											
4											
6											
8											
0											
2											
4											
6											
8											
0											
2											
4											
6											
8											
0											
2											
4											
6											
8											

Appendix K – Program development tools

ATARI MC68000 ASSEMBLERS

There are a number of assemblers available for the Atari ST programmer, they have small discrepancies in the assembly syntax used, no uniformity in the library and utility files supplied or of the method of creating an executable program.

This makes it difficult for the inexperienced programmer to type as source a program listing created for another assembler, and to get it operational. What I have tried to produce is an analysis of each assembler and a conversion chart that may help in isolating fairly straightforward problems.

Where a published program uses a particular assembler specific facility (special macros etc.) then translation will not always be possible by simple substitution and there may be no easy solution. Hopefully the general assembler compatibility chart will indicate whether there is the likelyhood of a conversion.

This guide is very much less than perfect, but it is an attempt at assisting inexperienced programmers in a very difficult field.

The assemblers

Very few of the assemblers provide programming details of the Motorola M68000 processor instruction set, or teach the user basic assembler language programming. If the reader has not written assembly language programs before, the brief overview of the language in Appendix G and H should help.

Seka

The combined editor/assembler/monitor/debugger is held in a very compact 20K of code, this means that parts of the package are a bit weak. Although two editors are supplied, a line editor and a screen editor, neither performs block find and replace function. Use the Atari wordprocessor in non wp mode for major or block changes in large files. What is likely to be more of a significant problem is the limitation to a leading letter for label and symbol names (the use of an underscore is very common in most libraries and the Atari system variables). A possible solution would be to substitute a little used letter for the leading underscore, say 'z'.

On the positive side, the Seka assembler generates absolute or relocatable executable code directly, has limited macro and conditional capability, and is a quick assembler for writing programs if you know what you are doing – some of the runtime error messages are incomprehensible with no guide in the manual as to what they are trying to tell you. It is very convenient having all the facilties in one program, an assembly error leaves the editor at the erroneous line for immediate correction and reassembly or the programmer may trace through the code with the monitor/debugger. The editor also allows the programmer to type the source code in free format; the assembled output listing is automatically tabulated, but there is no way of simply listing the code to a printer in a tabulated form which makes the source difficult to read when trying to debug program logic errors. Source files

entered in a tabulated form are occasionally detabulated in parts of the assembly list file. The system for linking files i. a bit messy and very non-standard as are some of the assembler directives. The monitor/debugger allows the programmer to single/multiple step through a program, examine registers, set breakpoints and provides all the necessary facilities to aid program debugging. It is important to ensure that program files are of even length; odd file lengths sometimes produce run-time errors not discovered by the debugger, which makes the fault extremely difficult to locate. The assembly syntax is pretty standard; labels must terminate in a colon, 'movea' should be entered as 'move', the assembler correcting the syntax but strangely 'adda' is acceptable. The 36 page manual limits the two examples to very simple TOS programs, one of which includes macros. The manual has a lot of ground to cover which it manages only at a fairly minimal level. i.e it does not provide enough information regarding the cause of errors – an error in a macro is flagged as an illegal operand in the calling code. The manual contains a very useful single page command summary.

The package is very easy to use, although not as powerful as some of the other packages in this Appendix. As an assembler, it is complete with minimal libraries of DOS calls equates and GEM array generators.

Hisoft

A combined editor/assembler with a seperate monitor/debugger. The Hisoft assembler employs include files to ease the access to the GEM and TOS functions and produces machine code directly. The include files require function parameters to be explicitly placed in the parameter arrays as per the assembler GEM example (Appendix L). The assembler package incorporates a selective GST compatible linker and does not like labels followed only by comments on the same line. The editor is a full feature program with the minor omission of displaying and handling only one file at a time.

The package contains include files of equates for BDOS, BIOS, extended BIOS calls, system variables and a GEM include file that provides program initialisation, VDI and AES constant equates and parameter array initialisation. The package does not provide details of the data (Chapter 3) to be placed in the arrays.

The monitor/debugger besides supporting the usual step, set breakpoints, examine and modify registers and memory etc. enables the assembled program to be run and debugged using seperate screens for the graphics and the monitor output, a very useful feature.

The documentation is well written and provides a good introduction for the beginner.

A very friendly package that could benefit from the use of a RAM disk to hold all the files in memory at once. In a 512K machine, seperating the program into two components does not appear to provide the optimum 'modus operandi'.

GST

The GST assembler package has a very good GEM based editor that enables up to four files to be worked on at the same time in multiple windows, copying blocks from one to another with ease. The only possible complaint regarding the editor could be the relatively slow loading of program and files and of cursor movement up and down within the file.

The assembler can produce relocatable binary output suitable for the linker or executable code directly from position independant source. The executable code does not contain the standard Atari TOS file header preamble, which must be added by the programmer if the file is to act as a stand alone program. A very useful list of the instruction mnemonics is provided, as is information on the optimisation route taken in compiling code. The use of an underscore for the leading character of a label or symbol is not permitted, which entails a degree of non compatibilty with the standard Atari ST notation for some system variables and extended BIOS calls.

The assembly of source is slow by virtue of the many disc accesses, but the use of RAM disc for compilation and the loading of all modules into memory together will obtain reasonable speed. There is no uninitialised data (BSS) d The GST linker is also supplied with the Metacomco macro assembler, it enables other high level language modules written in Pascal, Assembler and C to be linked together in a single program.

The library supplied contains macro definitions of conditional structures. GEM and TOS libraries are not supplied.

The documentation consists of seperate (unindexed) assembler, editor and linker manuals, which are very well packaged in a ring binder, the manuals are very detailed, but may be difficult for the inexperienced assembly language programmer to read.

Metacomco

The screen editor is good but does not follow the normal GEM style of access, although the user will very quickly adjust. The global 'find and replace' is comparitively slow as the screen is re-written for each change.

The assembler is slow in comparison with the smaller assemblers evaluated in this Appendix and would benefit greatly from the use of RAM disk. Symbols may be of up to thirty significant characters, but tabulated 'dc' data values on one source line seperated by a comma and space are not permitted – the space may be used to introduce a comment.

Metacomco supply the GST linker with their macro assembler which can produce either a binary file suitable for the GST linker or a CP/M 68K object file suitable for the DR link68 linker; which links to the complete DR set of GEM and TOS libraries, but is undocumented.

The Metacomco assembler package is supplied with assembler source to the GEM libraries and a monitor program. The monitor provides breakpoints, a trace mode and register/memory change and examine facilities.

The assembler suite of programs can be batched using the Menu+ program provided, It enables the sequence of edit, assemble, link and run to be controlled by the menu file, which runs and loads each program producing the specified outputs as requested and entering the next stage automatically via a pause, wait or continue programmed instruction.

The documentation is concise and very well laid out, but gives no additional explanation on assembly errors to that displayed on the screen during the assembly phase.

Digital Research

The Digital Research package can use any editor/word processor that is capable of producing an unformatted ASCII text file.

The assembler, which is a reasonable implementation of the Motorola M68000 assembly language, has no macro facilities but optimises instructions and branches to produce efficient code.

The DR LINK68 linker provides access to the DR GEM and TOS libraries that consist of accessory and application header files, GEMDOS, BIOS, XBIOS, VDI, AES and floating point libraries.

Although the assembler, linker and relocator programs can be installed as TTP (TOS Takes Parameters) files, the programs are much easier to run via the Activenture Corp. batch program. The additional use of a RAM disk to hold the files and programs produces a very reasonable response, eliminating much of the disc access.

The development package was intended for software developers and not the general public, as such it is written with a high degree of technical jargon. Complete with no omissions, a veritable 'War and Peace'. It provides information on all of the GEM VDI functions and makes no mention of those not implemented on the ST.

DR C language modules may also be linked with the assembled source and DR libraries to produce executable programs.

The package is supplied with the DR symbolic interactive debugger 'SID' enabling the program writer to test and debug M68000 executable code, either from TOS or GEM, read/write/move blocks of memory, disassemble code or produce a hex dump, examine the CPU state, trace, run or step through the code.

Compatibility table

The analysis of each package necessarily concentrates on the flaws, looking for inconsistences and omissions. What may not be apparent is how good in absolute terms the packages are, any purchaser being able to justify the cost on technical excellence alone.

It may be useful to give an indication as to the range of likely purchasers of each package:

Kseka assembler
Absolute beginner – competent programmer: very fast program development

Hisoft devpak
Absolute beginner – competent programmer: fast program development, with GEM and TOS bindings.

GST macro assembler
Absolute beginner – expert: full feature assembler capable of linking with other high level language modules to form executable programs.

Metacomco assembler
Competent programmer – expert: full feature assembler with macros. DR's linker may be used to provide access to the complete set of system libraries, and GST's linker to link high level language modules into a combined language program.

Digital Research assembler
Software developer: not available to the general public.

GENERAL ASSEMBLER COMPATIBILITY

Not exhaustive, merely a guide to what facilities are available.

Function	*Hisoft macro assembler*	*Digital Research*	*GST macro assembler*	*Metacomco macro assmbler*	*Kseka assembler*
Editor	GENST	–	Edit	Ed	All-in-one package
multifile edit	No	Can use any wordprocessor	Yes (4)	No	No
screen/line	screen		screen	screen	Line & screen
GEM	Yes	ASCII text.	Yes	No	No
Assembler(i/p)	(.S)	AS68 (.S)	(.ASM)	(.ASM)	(.S)
Output	Executable	Binary file	Binary file or executable (no file hdr)	See linker	Executable
no link	or binary				
Optimiser	Yes	Yes	Yes	Yes	
Macros	Yes	No	Yes	Yes	Yes
conditional	Yes	Yes	Yes	Yes	Yes
Linker	LINKST	Link68	GST-LINK	Can use either	Relocatable
Input			Binary file	GST-LINK or	mode
Submissions		Batch file	Control file	DR's LINK68	only
Output	GST compatible linker		File-header reloc tab code data opt symb tabl	and RELMOD programs (GST-LINK is supplied)	(odd format)
Libs					
GEM	Yes *(limited)*	Yes *(complete)*	No	Yes (source)	Yes *(minimal)*
TOS	Yes *(limited)*	Yes *(complete)*	No	No	Yes *(minimal)*
Maths	No	Yes	No	No	No
Monitor	Yes (MONST)	—	No	Yes	Yes
Debugger	Yes	SID symbolic interactive debugger	Not supplied Linker can put debug symbols in program	Supplied as a source example on disk.	Yes
Relocator prg	No	RELMOD	No	No	No
Symbols	16 sig chars		8 signif char	Upto 30 chars	
Label col 1	Space	Spc or colon	Spc or colon	Spc or colon	Colon
end col n	Space or :	Colon	Colon	Colon	Colon
Directives		Optional period			
Comment col 1	*	*	* or ;	* or ;	* or ;
col n	space	*	; or space	; or space	;
Case (Symbols)	Selectable	Significant	Not signifcnt	Selectable	Not signifcnt
Quotes	Single/double	Single/double	Single	Single/double	Single/double

GST assembler executable code must supply a TOS program header and be written in positition-independant code before it can be run.

Default file extensions are given in brackets.

ASSEMBLER DIRECTIVES COMPATIBILITY

Directive	*Explanation*	*Hisoft*	*Digital Research*	*GST*	*Metacomco*	*Kseka*
Include (i/p)	Insert external file	Yes (.S)	No	Yes (.IN) (.MAC)	Yes	Abs. code via linker
Text	Relocatable code	No	Yes	Section code	Yes (def)	Code = =
Data	Initialised data	No	Yes		Yes	No
BSS	Uninitialised data	DSBSS = =	Yes	No	Yes	Data = =
even	Align to word	Yes***	Yes	***	***	Yes & odd
ORG <addr>	Absolute section	Yes	Yes	Yes	No	Yes
Common	Common region	No	Yes	Yes	No	No
RORG <add>	Adjust curr locn	No		Yes	Yes	No
Offset	Define table via a DS directive	No	Yes	Yes	Yes	No
OPT	Select addr mode	Diff meaning	Ignored	PC or Abs	No	No
Globl	External label	No	Yes	No	No	Yes
Xref	External name	Yes	Yes	Yes	Yes	No
Xdef	Internal label for external use	Yes	Yes	Yes	Yes	No
Module	Lnk mod name	Yes		Yes		
Comment	Incl comments in linker listing			Yes		
Equ	Symbol	Yes	Yes	Yes	Yes	Yes & =
Equr	Register	Yes			Yes	
Reg	Register list		Yes	Yes	Yes	
Set	Temporary value	Yes	Uses Equ		Yes	
DC	Constant	Yes	Yes	Yes	Yes	Yes
DS	Storage	Yes	Yes	Yes	Yes	
DCB	Constant block			Yes	Yes	Blk = =
RS		Yes				No
Conditionals		Yes	Yes	Yes	Yes	Yes
IF eq,ne,gt ge,lt,le		Yes	Yes	No	Yes	If,Else IFB
String c,nc		Yes	Yes	Yes	Yes	No
Symbol d,nd		Yes		Yes	Yes	No
Library System		GEM/TOS	GEM/TOS/F P	None	None	Minimal
Macro				If,Else,For While,Until Repeat,Case		
	Sub nth arg	\n	Not	[n]	?n	\n
	Sub unique # nnn	\@	applicable	[.L]	?o	\@
Mask2			Ignored		Ignored	
IDNT			Ignored		Ignored	

*** DS and DC word and longwords automatically align to boundaries

ASSEMBLER CONVERSIONS

There are a number of assemblers available for the ST, each with different characteristics, this section is provided as an aid to translation of programs presented from alternative sources.

One of the by-products of the compatibility information is that it provides the opportunity of generating a subset of directives and instructions that are of almost universal applicability, but compatibility does tend to look at the lowest common denominator.

All of these programs have other attributes which provide a significant improvement in performance over the base standard, these improvements are not always apparent to the casual user but very handy to have if required.

If you wish to write source for maximum compatibility with other assemblers, the following should minimise the problems:

- Size all instructions (move, clr, lea etc. do not default)
- Size branches (avoids masses of GST warning messages)
- Avoid using reserved words for labels such as text, code etc.
- Use a semicolon for all comments (except Hisoft and DR which should use a '*')
- Do not tabulate DC data, added spaces do not travel well.
- Limit label and symbol lengths to eight characters.
- Use 'EQU' directive, not '='.
- After text and data sections, it is wise to ensure that the PC is on a word boundary. Most programs use 'EVEN', some assemblers use 'DC' and 'DS' with a .W or a .L extension.

Sizing instructions is perhaps the most difficult factor to come to terms with. I find it extremely difficult to ignore and not stop and read any warning messages, and become a little irritated to find that a branch 'might be short' or that LEA has not got a .L extension. All warnings should be significant or else they will all be treated as superfluous information.

General conversion chart

from ↓	*To Kseka*	*To Hisoft*	*To GST*	*To Metacomco*	*To DR*	*Comments*
Macros Kseka comment Opcodes	MACRO ?1 ?2 ?o ; blk	 → \1 → \2 → \@ → * → ds	MACRO A → [A] → [B] →[.L] Size opcodes & branch → ds	 → [\1 → \2 → \@ Size opcodes → ds	n.a. Expand code in full → * → ds	Program may use many labels that do not appear to have any function. They will probably be breakpoints.
Macros Hisoft comment Opcodes	 → ?1 → ?2 → ?o → ; → blk	MACRO \1 \2 \@ * ds	MACRO A → [A] → [B] →[.L] → ; Size opcodes & branch	 → ;	n.a. Expand code in full	Places GEM data directly into VDI and AES arrays. If the source includes GEM calls, follow the examples in Appendix L
Macros GST comment Opcodes	MACRO MACRO → ?1 → ?2 → ?o → blk	 → \1 → \2 → \@ → * ds	MACRO A [A] [B] [.L] ;	MACRO → \1 → \2 → \@	n.a. Expand code in full	Seems to like all instructions sized or issues lots of warning messages. Library of conditional macros will cause problems. Code has to belong to a section.
Macros Metacomco comment Opcodes	 → ?1 → ?2 → ?o → blk	 → *	MACRO A → [A] → [B] →[.L] Size branch	MACRO \1 \2 \@ ; ds	n.a. Expand code in full in appx L)	GEM lib supplied Implementation is standard but translation depends on the availability of the library used, (follow examples
Digital Research comment Opcodes	→code →data →zlabel → ; (add to all labels) → blk		Section C Section D →zlabel → ;	→ ;	TEXT BSS _label * ds	Delete any period directive prefix. No macro facilities but a full set of GEM libraries. Look at examples Appx L to see sheer power and how difficult translation will be

The above table will help to eliminate some of the more straight-forward program conversion problems, those that remain are likely to be due to the use of assembler specific directives and or libraries (especially label errors).

If a program is published, one assumes that any include file data will be generally available and can either be appended as an include file or the code integrated with the main program block of code.

If your assembler does not have VDI and AES libraries, then to use GEM you will have to create the arrays and load the addresses as shown in the assembler GEM example in Appendix L.

The chart is limited to simple conversions. Once include files and global label definitions are used, you will need to assemble the program, generate a list of the missing external labels and hopefully find them in the examples Appendix and/or the call listings Appendix E.

Numbers

The following shows the standard presentation of the various numeric types:

Binary	%xxxxxxxx	x = 0 or 1
Decimal	nnnn	n = 0 to 9
Hexadecimal	$nnnn	n = 0 to 9, a to f

BASIC CALLING PROCEDURES

These are for simple source files assembled (and linked) without libraries.

KSEKA

```
SEKA>r                    Instruction to read source file from disc
FILENAME>filename         File to read (default .S extension)
SEKA>a                    Instruction to assemble source
OPTIONS>v                 Option to view assembly on screen

SEKA>wo                   Instruction to write output program
FILENAME>filename         File to write (default .PRG extension)
```

HISOFT

Menu driven, place cursor over instruction and click

```
Option --> Assemble
                                    Option dialog box
Binary filename xxxxx.prg
Listing option boxes (none/screen/printer/disc)
Assemble/cancel boxes
```

GST

Menu driven, place cursor over instruction and click, *or* double click the TTP program file and enter the filename as the parameter:

`ASM.PRG` filename to produce a list file and `filename.BIN` from a default `.ASM` extension file

`LINK.PRG` filename to produce a `.MAP` file and `filename.PRG` from a default `.BIN` extension file

METACOMCO

The program files are installed as Tos Takes Parameters (TTP), double click and enter input file:

`ASSEM.PRG filename` assembles filename.asm to produce a GST format output file

`LINK.PRG filename` produces a `.MAP` file and `filename.PRG` from a default `.BIN` extension file

DIGITAL RESEARCH

The program files are installed as Tos Takes Parameters and Research the file data entered into the parameter box.

```
AS68.PRG filename.S                    Produce binary file
LINK68.PRG filename.68K = filename.o   Produce relocatable file
RELMOD.PRG filename                    Produce absolute file
```


Executable file sizes (bytes)

Natural compilations with no optimisation extensions called

Program	*Page #*	*D.R*	*Seka*	*Hisoft*	*GST*	*Metacomco*
GEM error message	L7	777	–	–	–	781
Assembler GEM	L8-L17	1651	1734 3170	3170	3016	3170
TOS colour demo	L16	145	162 246	246	235	248
TOS VT52 screen	L18-19	194	202	202	192	202
TOS sound program	L20-22	324	331 591	591	586	591
A-line sprite prog	L25-26	296	314 394	394	374	392

The Metacomco file sizes are for files linked via the GST linker except for the 'GEM error message' which used the DR linker.

The Digital Research files are absolute files and therefore presumably nearer the minimum possible size.

(a secondary figure is given which specifically sets all text and data sections initialised to provide standard file size compilation comparisons, not all of the test assemblers can handle an uninitialised section)

C COMPILERS

Many C compilers have been developed for the ST range of computers, enabling the ST programmer to produce modular, well documented, easily maintained code that may be ported to other C systems with a minimum of effort.

Although achieving the same end result, the C compilers differ considerably in the way that they attain that result. I give three examles of the compilation process:

DIGITAL RESEARCH C

Source text

↓ [Preprocessor] — *Resolve define and include statements*

Intermediate file

↓ [Parser] — *Produce an intermediate code file*

Intermediate code file

↓ [Code generator] — *Generate assembly language source file*

Assembler text

↓ [Asembler] — *Create an object file*

Object file

↓ Libraries

↓ [Link68] — *Link object file with run-time library and operating system files*

Executable program file

↓ [Relocator] — *Change the relocatable information to absolute data*

Absolute program file

GSTC COMPILER

Source text

↓ [C compiler] — *Resolve, define and include statements, produce assembly source file.*

Assembler text

↓ [Assembler] — *Produce a relocatable binary file*

Relocatable binary file

↓ Libraries

↓ [Link68] — *Link object file with run-time library and operating system files*

Executable program file

METACOMCO LATTICE C

Source text

↓ [LC1 compiler] — *Produce an intermediate file of logical records*

Intermediate quad file

↓ [LC2 compiler] — *Produce either an object file (DR linker) or a relocatable binary file (GST linker)*

Binary file (GST or TOS format)

↓ Libraries

↓ [Link68] — *Link object file with run-time library and operating system files*

Executable program file

Those programmers who wish to program the Atari ST in C may find the following brief notes helpful.

Unlike nearly all of the commercial assemblers, the C compilers supply a full set of GEM and system libraries. Commercial C compilers for the Atari ST will, in general, adhere to the GEM VDI and GEM AES function names used in this book, usually only the first 8 characters being significant. The compilers diverge

considerably in use of parameter names, the call and parameters are therefore not provided here to avoid confusion. The manual of the C compiler you are using will provide a definitive list of the library routines available, interfaces, and the required parameter size, sequence and annotation.

Many compilers will have additional features to greatly simplify the task of writing GEM programs; but bear in mind that the use of these 'super C' GEM functions may put a restraint on program portability. A definite decision should be made as to whether the programmer is writing portable code or simply writing a program.

On a more general note; it is very much easier to develop programs on a 1 Megabyte disk drive, which makes the larger drive well worth the small additional cost for those machines without the built-in drive.

Appendix L – Example assembler programs

The programs presented in this section illustrate some of the techniques involved in accessing parts of the Atari ST operating system and also present general purpose header/include files. The programs are written as shells to which the programmer may add his/her own composition.

It is not the intention to provide 'state of the art' programs, merely demonstrate access to the various parts of the operating system. Any attempt at definitive programming would rapidly succumb to the passage of time and tend to produce a book of listings. The main place to find quality programs will be the computer magazines, where programs developed from this and other books will appear as programmers quickly find new, smarter routes to access and use the ROM routines.

Desktop accessories should be compiled as applications for debugging purposes as it is not possible to execute an accessory.

Program conversion key:

n.a	program not suitable for this assembler.
xxx	delete this line
*	use ; for Kseka, GST and Metacomco comments

GEM

APPLICATION AND ACCESSORY HEADER FILE

Digital Research (and Metacomco in CP/M 68K object mode) application and desk accessory files require a similar type of header source file construct to provide access to the GEM VDI and AES libraries, either as the first file in the DR link statement or as the beginning of a single block of assembler code.

Part of this file determines the size of memory the application requires and returns the remainder to GEMDOS. Some Atari ST assemblers will provide similar code as a header/initialisation file to permit the programmer to access the VDI and AES functions through their own integral libraries.

```
* Digital Research                        * Hisoft .  GST  Metacomco  Seka
*                                         *  n.a  .  n.a  .        .  n.a  .
  text              * Text segment        *       .       .        .       .
*
  globl  _main      * Make labels         *       .        . xdef  .       .
  globl  _crystal   * accessible to       *       .        . xdef  .       .
  globl  _ctrl_cnts * external files      *       .        . xdef  .       .
*
  move.l a7,a5      * store stack (a5)    *       .        .       .       .
  move.l #ustk,a7   * set local stack     *       .        .       .       .
*
* Desk accessories do not require the following lines of code
*  which size memory and return the unused memory to GEMDOS
*
  move.l 4(a5),a5   * basepage address    *       .        .       .       .
  move.l $c(a5),d0  * length of text      *       .        .       .       .
  add.l  $14(a5),d0 * length of data      *       .        .       .       .
  add.l  $1c(a5),d0 * length of BSS       *       .        .       .       .
  add.l  #$100,d0   * basepage size       *       .        .       .       .
  move.l d0,-(sp)   * retained mem len    *       .        .       .       .
  move.l a5,-(sp)   * memory to modify    *       .        .       .       .
  move   d0,-(sp)   * dummy word          *       .        .       .       .
  move   #$4a,-(sp) * reallocate to GEM   *       .        .       .       .
  trap   #1         * function number     *       .        .       .       .
  add.l  #12,sp     * tidy stack          *       .        .       .       .
*
* Main program call
*
  jsr    _main      * main program code   *       .        .       .       .
  move.l #0,-(a7)   * return to GEMDOS    *       .        .       .       .
  trap   #1         * function call       *       .        .       .       .
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

* GEMAES calls link through _crystal
*
_crystal:
  move.l 4(a7),d1 * address of AES pblk*
  move.w #200,d0  * GEMAES function #  *
  trap   #2       * function call     *
  rts             * return            *
*
  bss             * block storage seg *
  even            * force even boundary*
*
        ds.l  64                      *
ustk:   ds.l  1                       *
*
  data                                *
  even                                *
*
_ctrl_cnts
*
*  Application manager
*
  dc.b  0,1,0   * APPL_INI.......10   *
  dc.b  2,1,1   * APPL_REAd......11   *
  dc.b  2,1,1   * APPL_WRIte.....12   *
  dc.b  0,1,1   * APPL_FINd......13   *
  dc.b  2,1,1   * APPL_TPLay.....14   *
  dc.b  1,1,1   * APPL_TREcord...15   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,1,0   * APPL_EXIt......19   *
*
*  Event manager
*
  dc.b  0,1,0   * EVNT_KEY.......20   *
  dc.b  3,5,0   * EVNT_BUTton....21   *
  dc.b  5,5,0   * EVNT_MOUse.....22   *
  dc.b  0,1,1   * EVNT_MESsage...23   *
  dc.b  2,1,0   * EVNT_TIMe......24   *
  dc.b  16,7,1  * EVNT_MULti.....25   *
  dc.b  2,1,0   * EVNT_DCLick....26   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
```



```
* Digital Research                    * Hisoft .  GST  Metacomco  Seka

*  Menu manager
*
  dc.b  1,1,1   * MENU_BAR.......30   *
  dc.b  2,1,1   * MENU_ICHeck....31   *
  dc.b  2,1,1   * MENU_IENable...32   *
  dc.b  2,1,1   * MENU_TNOrmal...33   *
  dc.b  1,1,2   * MENU_TEXt......34   *
  dc.b  1,1,1   * MENU_REGister..35   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
*  Object manager
*
  dc.b  2,1,1   * OBJC_ADD.......40   *
  dc.b  1,1,1   * OBJC_DELete....41   *
  dc.b  6,1,1   * OBJC_DRAw......42   *
  dc.b  4,1,1   * OBJC_FINd......43   *
  dc.b  1,3,1   * OBJC_OFFset....44   *
  dc.b  2,1,1   * OBJC_ORDer.....45   *
  dc.b  4,2,1   * OBJC_EDIt......46   *
  dc.b  8,1,1   * OBJC_CHAnge....47   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
*  Form manager
*
  dc.b  1,1,1   * FORM_DO........50   *
  dc.b  9,1,1   * FORM_DIAlog....51   *
  dc.b  1,1,1   * FORM_ALErt.....52   *
  dc.b  1,1,0   * FORM_ERRor.....53   *
  dc.b  0,5,1   * FORM_CENtre....54   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
```

```
* Digital Research                    * Hisoft . GST Metacomco Seka

*  Dialog manager
*
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
*  Graphics manager
*
  dc.b  4,3,0   * GRAF_RUBberbox.70   *
  dc.b  8,3,0   * GRAF_DRAgbox...71   *
  dc.b  6,1,0   * GRAF_MOVebox...72   *
  dc.b  8,1,0   * GRAF_GROwbox...73   *
  dc.b  8,1,0   * GRAF_SHRinkbox.74   *
  dc.b  4,1,1   * GRAF_WATchbox..75   *
  dc.b  3,1,1   * GRAF_SLIdebox..76   *
  dc.b  0,5,0   * GRAF_HANdle....77   *
  dc.b  1,1,1   * GRAF_MOUse.....78   *
  dc.b  0,5,0   * GRAF_MKState...79   *
*
*  Scrap manager
*
  dc.b  0,1,1   * SCRP_REAd......80   *
  dc.b  0,1,1   * SCRP_WRIte.....81   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
```



```
* Digital Research                    * Hisoft . GST Metacomco Seka
*  File selector manager
*
  dc.b  0,2,2   * FSEL_INPut.....90   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
*  Window manager
*
  dc.b  5,1,0   * WIND_CREate...100   *
  dc.b  5,1,0   * WIND_OPEn.....101   *
  dc.b  1,1,0   * WIND_CLOse....102   *
  dc.b  1,1,0   * WIND_DELete...103   *
  dc.b  2,5,0   * WIND_GET......104   *
  dc.b  6,1,0   * WIND_SET......105   *
  dc.b  2,1,0   * WIND_FINd.....106   *
  dc.b  1,1,0   * WIND_UPDate...107   *
  dc.b  6,5,0   * WIND_CALc.....108   *
  dc.b  0,0,0   *                     *
*
*  Resource manager
*
  dc.b  0,1,1   * RSRC_LOAd.....110   *
  dc.b  0,1,0   * RSRC_FREe.....111   *
  dc.b  2,1,0   * RSRC_GADdress.112   *
  dc.b  2,1,1   * RSRC_SADdress.113   *
  dc.b  1,1,1   * RSRC_OBFix....114   *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
  dc.b  0,0,0   *                     *
*
*  Shell manager
*
  dc.b  0,1,2   * SHEL_REAd.....120   *
  dc.b  3,1,2   * SHEL_WRIte....121   *
  dc.b  1,1,1   *                     *
  dc.b  1,1,1   *                     *
  dc.b  0,1,1   * SHEL_FINd.....124   *
  dc.b  0,1,2   * SHEL_ENVrn....125   *
*                                     *
    end
```

The object file is used as the first file in the link to produce an Atari ST program file, say myprog, that accesses the DR GEM libraries i.e:

either ***DR***

```
as68 -l -u apstart.s
```

or ***Metacomco***

```
assem.prg apstart.asm opt j
```

followed by the linking of the main program file (see following example) to the header and the DR library files.

```
link68 [u] myprog.68=apstart myprog.o vdibind aesbind
```

and finally relocated using:

```
relmod myprog
```

Delete all temporary files, leaving either an application file `myprog.prg` (which may be run by double clicking the icon in the directory listing) or an accessory file which must be renamed myprog.acc. Reboot the system and run the file by clicking the icon in the list of 'Desk' accessory files.

Remember to initially compile and run accessories as applications to debug them.

GEM DEMONSTRATION PROGRAM

To use GEM directly, push the function parameters onto the stack in the order given by the GEM VDI and GEM AES tables, ensuring that the correct size of parameter is pushed.

The following program, which may be written in either DR or Metacomco macro assembler but must use the DR link68 linker, lists in descending order the TOS error codes in dialog boxes, the user stepping from one code to the next via the mouse or the 'return' key.

```
* Digital Research                          * Hisoft   GST  Metacomco  Seka
*                                           *  n.a  .  n.a  .        .  n.a  .
* Demo GEM program
*
    globl  _main      *                     *      .        . xdef  .      .
    globl  _form_err  *                     *      .        . xref  .      .
    globl  _appl_ini  *                     *      .        . xref  .      .
    globl  _appl_exi  *                     *      .        . xref  .      .
*
    text                                    *      .        .  The  .      .
*                                           *      .       Metacomco       .
_main:                                      *      .       .external       .
    jsr    _appl_ini                        *      .       .symbol .       .
*                                           *      .        . names .      .
    move.w  #63,d4    * Error start #       *      .        . are   .      .
loop: move.w  d4,temp * Save it             *      .       .limited.       .
    move.w  d4,-(sp)  * Stack it            *      .        . to 8  .      .
    jsr    _form_err * What is it           *      .       characters      .
    add.w  #2,sp      * Tidy it             *      .         i.e   .       .
    move.w  temp,d4   * Recover it          *      .        .      .       .
    dbra   d4,loop    * and next            *      .       _form_er        .
    jsr    _appl_exi * Controlled exit      *      .       _appl_in        .
    rts                                     *      .       _appl_ex        .
*
    bss                                     *      .        .      .       .
*
temp:  ds.w  1                              *      .        .      .       .
*
    end
```

The file may be assembled using:

either ***DR***

```
as68 -l -u -p myprog.s
```

or ***Metacomco***

```
assem myprog.asm opt j
```

Both programs are linked with the Digital Research link68 linker i.e:

```
link68 [s,u] myprog.68k=apstart,myprog.o,aesbind
```

Finally relocate using:

```
relmod myprog
```


GEM DEMONSTRATION ASSEMBLY PROGRAM

It is possible to write assembly language programs that do not use the DR GEM bindings but simply access the functions via the Extended BDOS TRAP #2 calls. The following example shell shows a technique that will enable the programmer to create a window, do some work in it, and then make a controlled exit.

Note: Although the window is created with the sizing diamond and sliders, no code has been written to handle the screen managers requests for change; if these functions are activated they are ignored. If the cursor is active (as in this program) and covers part of the foreground content of the screen when the program is loaded, it will leave a hole when moved.

```
* Digital Research                          * Hisoft   GST  Metacomco  Seka
*
*    Assembler GEM program
*
*  Size the job and free back to GEMDOS unused memory
*
*
   text                                     *  xxx  section c        . code  .
*
   move.l  a7,a5        * curr  -> a5   *
   move.l  #ustk,a7     * set local stk *
   move.l  4(a5),a5     * get base page *
   move.l  $c(a5),d0    * text segment  *
   add.l   $14(a5),d0   * data segment  *
   add.l   $1c(a5),d0   * uninitialized *
   add.l   #$100,d0     * basepage size *
   move.l  d0,-(sp)     *               *
   move.l  a5,-(sp)     *               *
   move    d0,-(sp)     *               *
   move    #$4a,-(sp)   *free unused mem*
   trap    #1           *               *
   add.l   #$c,sp       * tidy stack    *
   jsr     start        *               *
   move.l  #$0,-(sp)    * ret to GEMDOS *
   trap    #$1          *               *
*
*  Technique for setting up VDI & AES arrays
*
*     Initialize AES arrays
*
start:
*
   jsr     ini_aes      *               *
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka
*  Call APPL_INI (1st call)  see section 3
*
appl_ini:
   move.w  #$a,control                 *
   move.w  #$0,control+2               *
   move.w  #$1,control+4               *
   move.w  #$0,control+6               *
   jsr     aes                         *
   tst.w   int_ou                      *
   bpl     graf_han                    *
   rts                                 *
*
*  Call GRAF_HAN to get name of the currently opened window.
*
graf_han:
   move.w  #77,control                 *
   move.w  #$0,control+2               *
   move.w  #$5,control+4               *
   move.w  #$0,control+6               *
   jsr     aes                         *
   move.w  int_ou,handle               *
*
*  Initialize VDI arrays
*
   jsr     ini_vdi                     *
*
*  Open virtual workstation
*
v_opnvwk:
   move.w  #100,contrl                 *
   move.w  #0,contrl+2                 *
   move.w  #11,contrl+6                *
   move.w  handle,contrl+12            *
*
*  11 input parameters
*
   move.w  #1,intin     *drive id      *
   move.w  #1,intin+2   *line type     *
   move.w  #1,intin+4   *line color    *
   move.w  #1,intin+6   *marker type   *
   move.w  #1,intin+8   *marker color  *
   move.w  #1,intin+10  *text face     *
   move.w  #1,intin+12  *text color    *
   move.w  #1,intin+14  *interior fill *
   move.w  #1,intin+16  *fill index    *
   move.w  #1,intin+18  *fill color    *
   move.w  #2,intin+20  *NDC/RC        *
   jsr     vdi                         *
*
```



```
* Digital Research                    * Hisoft   GST  Metacomco  Seka
*  Save virtual screen workstation device handle
*
   move.w  contrl+12,vhandl            *
   tst.w   contrl+12                   *
   beq     appl_exi                    *
*
* Test here for screen resolution and number of colors available
* (even in mono). Load appropriate resource file using the AES
* RSRC_LOA call if necessary.
*
*  Get max possible size of window
*
max_wind:
   move.w  vhandl,int_in               *
   move.w  #7,int_in+2          * sizes *
   jsr     wind_get                    *
   tst.w   int_ou                      *
   beq     appl_exi                    *
*
*  Calculate work area of window
*
   move.w  #0,int_in                   *
   jsr     wind_cal                    *
   tst.w   int_ou                      *
   beq     appl_exi                    *
*
*  Calc new window bordered area
*
   move.w  #1,int_in                   *
   jsr     wind_cal                    *
   tst.w   int_ou                      *
   beq     appl_exi                    *
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

*  Alloc space for full size window *
*
wind_cre:
   move.w  #100,control                *      .      .      .      .
   move.w  #$5,control+2               *      .      .      .      .
   move.w  #$1,control+4               *      .      .      .      .
   move.w  #$0,control+6               *      .      .      .      .
*
   move.w  #$0fff,int_in        * edges *     .      .      .      .
   move.w  int_ou+2,int_in+2    * x1    *     .      .      .      .
   move.w  int_ou+4,int_in+4    * y1    *     .      .      .      .
   move.w  int_ou+6,int_in+6    * x2    *     .      .      .      .
   move.w  int_ou+8,int_in+8    * y2    *     .      .      .      .
   jsr     aes                         *     .      .      .      .
*
   move.w  int_ou,whandl               *     .      .      .      .
   tst.w   int_ou                      *     .      .      .      .
   beq     appl_exi                    *     .      .      .      .
*
*  Open window at last  *
*
wind_ope:
   move.w  #101,control                *     .      .      .      .
   move.w  #$5,control+2               *     .      .      .      .
   move.w  #$1,control+4               *     .      .      .      .
   move.w  #$0,control+6               *     .      .      .      .
*
*  Absolute parameters
*
   move.w  whandl,int_in               *     .      .      .      .
   move.w  #0,int_in+2       * x1      *     .      .      .      .
   move.w  #0,int_in+4       * y1      *     .      .      .      .
   move.w  #280,int_in+6     * x2      *     .      .      .      .
   move.w  #160,int_in+8     * y2      *     .      .      .      .
   jst     aes                         *     .      .      .      .
   move    .      .      .      .
   tst.w   int_ou                      *     .      .      .      .
   beq     appl_exi                    *     .      .      .      .
*
*  Do something on the screen, this is where your program starts.
*
```



```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

*  Set screen parameters *
*
vsf_inte:
   move.w  #23,contrl                  *     .      .      .      .
   move.w  #0,contrl+2                 *     .      .      .      .
   move.w  #1,contrl+6                 *     .      .      .      .
   move.w  whandl,contrl+12            *     .      .      .      .
*
   move.w  #1,intin       * solid      *     .      .      .      .
   jsr     vdi                         *     .      .      .      .
*
*  Style                 *
*
vsf_styl:
   move.w  #24,contrl                  *     .      .      .      .
   move.w  #0,contrl+2                 *     .      .      .      .
   move.w  #1,contrl+6                 *     .      .      .      .
   move.w  whandl,contrl+12            *     .      .      .      .
*
   move.w  #1,intin       * n.a        *     .      .      .      .
   jsr     vdi                         *     .      .      .      .
*
*  Colour                *
*
vsf_colo:
   move.w  #25,contrl                  *     .      .      .      .
   move.w  #0,contrl+2                 *     .      .      .      .
   move.w  #1,contrl+6                 *     .      .      .      .
   move.w  whandl,contrl+12            *     .      .      .      .
*
   move.w  #1,intin       * black      *     .      .      .      .
   jsr     vdi                         *     .      .      .      .
*
*  Set mouse style       *
*
graf_mou:
   move.w  #78,control                 *     .      .      .      .
   move.w  #$1,control+2               *     .      .      .      .
   move.w  #$1,control+4               *     .      .      .      .
   move.w  #$1,control+6               *     .      .      .      .
*
   move.w  #$0,int_in                  *     .      .      .      .
   jsr     aes                         *     .      .      .      .
   tst.w   int_ou                      *     .      .      .      .
   beq     appl_exi                    *     .      .      .      .
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

*  Get position of window work area
*
where:
   move.w  whandl,int_in              *
   move.w  #4,int_in+2    * work area *
   jsr     wind_get                   *
   tst.w   int_ou                     *
   beq     appl_exi
*
*  Get coordinates within work area
*
   add.w  #35,int_ou+2                *
   add.w  #35,int_ou+4                *
   sub.w  #50,int_ou+6                *
   sub.w  #50,int_ou+8                *
*
*  Draw a shape from those coords *
*
v_rfbox:
   move.w  #11,contrl                 *
   move.w  #2,contrl+2                *
   move.w  #0,contrl+6                *
   move.w  #9,contrl+10               *
   move.w  whandl,contrl+12           *
*
*  Absolute coords -- not window the reason for this patch
*
   move.w  int_ou+2,ptsin             *
   move.w  int_ou+4,ptsin+2           *
   move.w  int_ou+6,ptsin+4           *
   move.w  int_ou+8,ptsin+6           *
*
   jsr     vdi                        *
*
*  Wait for a sign - about 1 minute *
*
evnt_tim:
   move.w  #24,control                *
   move.w  #$2,control+2              *
   move.w  #$1,control+4              *
   move.w  #$0,control+6              *
*
   move.w  #$ffff,int_in    *Lo       *
   move.w  #$0000,int_in+2 *Hi        *
   jsr     aes                        *
*
*  End of program, shut the window in a controlled manner
*
```



```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

*  Close v_scrn    Stop o/p (Shut window down) *
*
v_clsvwk:
   move.w  #101,contrl                *
   move.w  #0,contrl+2                *
   move.w  #0,contrl+6                *
   move.w  vhandl,contrl+12           *
*
*  Close window          *
*
wind_clo:
   move.w  #102,control               *
   move.w  #$1,control+2              *
   move.w  #$1,control+4              *
   move.w  #$0,control+6              *
*
   move.w  whandl,int_in              *
   jsr     aes                        *
   tst.w   int_ou                     *
   beq     appl_exi                   *
*
*   Deallocate space and handle *
*
wind_del:
   move.w  #103,control               *
   move.w  #$1,control+2              *
   move.w  #$1,control+4              *
   move.w  #$0,control+6              *
*
   move.w  whandl,int_in              *
   jsr     aes                        *
   tst.w   int_ou                     *
*  beq     appl_exi                   *
*
*  Call APPL_EXI (Last call) *
*
appl_exi:
   move.w  #19,control                *
   move.w  #$0,control+2              *
   move.w  #$1,control+4              *
   move.w  #$0,control+6              *
   jsr     aes                        *
   move.w  int_ou,d0                  *
   rts                                *
*
*  Subroutines
*
```

```
* Digital Research                          * Hisoft   GST  Metacomco  Seka

*  Get window data  *
*
wind_get:
   move.w  #104,control                      *      .      .      .      .
   move.w  #$2,control+2                     *      .      .      .      .
   move.w  #$5,control+4                     *      .      .      .      .
   move.w  #$0,control+6                     *      .      .      .      .
*
*  move.w  vhandl,int_in      * rem         *      .      .      .      .
*  move.w  #7,int_in+2        * out         *      .      .      .      .
   jsr     aes                               *      .      .      .      .
   rts                                       *      .      .      .      .
*
*  Calculate window work area based on facilities: title,
*   scroll bar etc.    *
*
wind_cal:
   move.w  #108,control                      *      .      .      .      .
   move.w  #$6,control+2                     *      .      .      .      .
   move.w  #$5,control+4                     *      .      .      .      .
   move.w  #$0,control+6                     *      .      .      .      .
*
*  move.w  #0,int_in          * rem out     *      .      .      .      .
   move.w  #$0fff,int_in+2    * edges       *      .      .      .      .
   move.w  int_ou+2,int_in+4  * x1          *      .      .      .      .
   move.w  int_ou+4,int_in+6  * y1          *      .      .      .      .
   move.w  int_ou+6,int_in+8  * x2          *      .      .      .      .
   move.w  int_ou+8,int_in+10 *y2           *      .      .      .      .
   jsr     aes                               *      .      .      .      .
   rts                                       *      .      .      .      .
*
*  VDI parameter block call
*
vdi:
   move.l  #contrl,pblock     * reset       *      .      .      .      .
   move.l  #pblock,d1                       *      .      .      .      .
   move.l  #115,d0                          *      .      .      .      .
   trap    #$2                              *      .      .      .      .
   rts                                      *      .      .      .      .
*
*  AES parameter block call
*
aes:
   move.l  #control,_c        * reset       *      .     zc      .     zc
   move.l  #_c,d1                           *      .    #zc      .    #zc
   move.l  #200,d0                          *      .      .      .      .
   trap    #$2                              *      .      .      .      .
   rts                                      *      .      .      .      .
*
```



```
* Digital Research                          * Hisoft   GST  Metacomco  Seka

*  Set up AES array
*
ini_aes:
   move.l  #control,_c                       *      . zc      .      . zc      .
   move.l  #global,_c+4                      *      . zc+4    .      . zc+4    .
   move.l  #int_in,_c+8                      *      . zc+8    .      . zc+8    .
   move.l  #int_ou,_c+12                     *      . zc+12   .      . zc+12   .
   move.l  #addr_in,_c+16                    *      . zc+16   .      . zc+16   .
   move.l  #addr_ou,_c+20                    *      . zc+20   .      . zc+20   .
   rts                                       *      .      .      .      .
*
*  Set up VDI array
*
ini_vdi:
   move.l  #contrl,pblock                    *      .      .      .      .
   move.l  #intin,pblock+4                   *      .      .      .      .
   move.l  #ptsin,pblock+8                   *      .      .      .      .
   move.l  #intout,pblock+12                 *      .      .      .      .
   move.l  #ptsout,pblock+16                 *      .      .      .      .
   rts                                       *      .      .      .      .
*
*    Make space for the arrays. You must ensure these are large
* enough to hold the array's data. Be especially careful regarding
* the spelling of the array names.
*
   bss                                       * xxx  section d       .  data .
   even                                      * xxx  .  xxx  .  xxx  .  xxx  .
*
        ds.l    256                          *      .      .      . blk.l .
ustk:   ds.l    1                            *      .      .      . blk.l .
*
pblock: ds.l    5                            *      .      .      . blk.l .
contrl: ds.w    12                           *      .      .      . blk.w .
intin:  ds.w    30                           *      .      .      . blk.w .
ptsin:  ds.w    30                           *      .      .      . blk.w .
intout: ds.w    45                           *      .      .      . blk.w .
ptsout: ds.w    12                           *      .      .      . blk.w .
*
handle: ds.w    1                            *      .      .      . blk.w .
vhandl: ds.w    1                            *      .      .      . blk.w .
whandl: ds.w    1                            *      .      .      . blk.w .
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

_c:       ds.l    6                   *        .  zc    .       .zc blk.l
control:  ds.w    5                   *        .        .       . blk.w .
global:   ds.w    16                  *        .        .       . blk.w .
int_in:   ds.w    16                  *        .        .       . blk.w .
int_ou:   ds.w    7                   *        .        .       . blk.w .
addr_in:  ds.l    2                   *        .        .       . blk.l .
addr_ou:  ds.l    1                   *        .        .       . blk.l .
*
  end
```

The program may be assembled and linked (if required) using the assembler calling procedures outlined in Appendix K.

TOS

DISPLAY DEMONSTRATION PROGRAM

The following program shows a typical Atari TOS file, which simply inverts the current mono display color for those programmers who,like myself, prefer white on black or toggles the border of a color display.

```
* Digital Research                    * Hisoft   GST  .  Meta .  Seka
*
*       Demo Atari TOS program
*
  text                                * xxx  section c        .  code .
*
  move.l  a7,a5                       *        .        .       .
  move.l  #ustk,a7    * set -> a7     *        .        .       .
  move.l  4(a5),a5                    *        .        .       .
  move.l  $c(a5),d0                   *        .        .       .
  add.l   $14(a5),d0                  *        .        .       .
  add.l   $1c(a5),d0                  *        .        .       .
  add.l   #$100,d0                    *        .        .       .
  move.l  d0,-(sp)                    *        .        .       .
  move.l  a5,-(sp)                    *        .        .       .
  move    d0,-(sp)                    *        .        .       .
  move    #$4a,-(sp)  * free unused   *        .        .       .
  trap    #1          * back to GEM   *        .        .       .
  add.l   #$c,sp                      *        .        .       .
  jsr     start       * jump to prg   *        .        .       .
  move.l  #$0,-(sp)   * terminate     *        .        .       .
  trap    #$1                         *        .        .       .
*
start:
  clr.l   -(sp)                       *        .        .       .
  move.w  #32,-(sp)   * set super     *        .        .       .
  trap    #1                          *        .        .       .
  move.l  d0,a1                       *        .        .       .
  move.w  #-1,d0      * get/set col   *        .        .       .
  jsr     newcol                      *        .        .       .
  eori    #1,d0                       *        .        .       .
  jsr     newcol                      *        .        .       .
exit:
  move.l  a1,-(sp)                    *        .        .       .
  move.w  #32,-(sp)   * set user      *        .        .       .
  trap #1                             *        .        .       .
  move.l  #0,-(a7)                    *        .        .       .
  trap    #1                          *        .        .       .
*
```

```
* Digital Research                    * Hisoft   GST  .  Meta .  Seka
newcol:
   move.w  d0,-(sp)   * get color     *      .      .      .      .
   move.w  #0,-(sp)                   *      .      .      .      .
   move.w  #7,-(sp)                   *      .      .      .      .
   trap    #14                        *      .      .      .      .
   add.w   #6,sp                      *      .      .      .      .
   rts                                *      .      .      .      .
*
   bss                                *  xxx  section d       . data .
        ds.l  20                      *      .      .      . blk.l .
ustk:   ds.l  1                       *      .      .      . blk.l .
*
   end
```

The above program is assembled and linked without any other files.

TOS HEADER FILE

The following shows a typical Atari TOS header file that may be incorporated in a user-written program to provide access to the base page offset variables.

```
*****************************************************
*
*     Base page format initialised by BDOS
*
*****************************************************
*
ltpa      equ 0          * Low TPA address
htpa      equ 4          * High TPA address + 1
lcode     equ 8          * Text segment start
codelen   equ 12         * Length of text segment
ldata     equ 16         * Initialized data segment start
datalen   equ 20         * Length of initialized data
lbss      equ 24         * BSS segment start
bsslen    equ 28         * Length of uninitialized data
*
*            either GEMDOS
*
*dta      equ 32         * pointer to DTA address
*parent   equ 36         * pointer to parents base page
*         equ 40         * reserved
*env      equ 44         * Environment string pntr (GEMDOS)
*
*             or Atari OS
*
freelen   equ 32         * Free memory length after BSS
ldriv     equ 36         * Drive from which program loaded
resvd     equ 37         * Reserved
fcb2      equ 56         * 2nd parsed fcb
fcb1      equ 92         * 1st parsed fcb
*
*            common tail
*
command   equ 128        * Command tail
```

Although it is good practice to size the memory requirements of your program and return the unused memory to GEMDOS, programs can be written without if they return to the GEM desktop.

GEM allocates all the memory to the program, only if it multitasks or calls and loads another program is there any real need to return spare memory.

CHARACTER PRINTING PROGRAM

This simple program demonstrates some of the methods available for printing to the VT52 screen. The compiled program may be installed as:

A 'GEM program' – the busy bee cursor will appear in the display and leave a hole when moved.

A 'TOS program' – with flashing cursor, the cursor may be hidden quite easily by incorporating within the prdat string the 'hide cursor' escape code as specified in Appendix C.

```
* Digital Research                          * Hisoft   GST  Metacomco  Seka
*
* Monochrome TOS VT52 screen print program (0,0 to 24,79)
*-----------------------------------------------------------
*    (For colour, set max print width in 'prdat' to 39)
*
   text                                     *  xxx  section c       .  code .
*          GEM BIOS type print
*
   clr.l   d4          * Clear d4          *        .        .        .
   clr.l   d5          * Clear d5          *        .        .        .
   lea     prdat,a4    * Get data address*        .        .        .
   move.b  (a4),d4     * Data count-1      *        .        .        .
cloop:
   adda.l  #1,a4       * Get next          *        .        .        .
   move.b  (a4),d5     * Getchar byte      *        .        .        .
   move.w  d5,-(sp)    * Stack char.w      *        .        .        .
   move.w  #2,-(sp)    * Send to console *        .        .        .
   move.w  #3,-(sp)    *Set bconout()      *        .        .        .
   trap    #13         * Call it           *        .        .        .
   add.w   #6,sp       * Tidy stack        *        .        .        .
   dbra    d4,cloop    * loop for next     *        .        .        .
*
*          GEM BDOS type print
*
   lea     mess,a0     * GetASCII string *        .        .        .
   move.l  a0,-(sp)    * Stack it          *        .        .        .
   move.w  #9,-(sp)    * set conws()       *        .        .        .
   trap    #1          * Call it           *        .        .        .
   add.w   #6,sp       * Tidy stack        *        .        .        .
*
   move.l  #200,d1     *                   *        .        .        .
exlp:
   move.l  #-1,d0      * Wait              *        .        .        .
dloop:
   dbra    d0,dloop    *                   *        .        .        .
   dbra    d1,exlp     *                   *        .        .        .
*
```



```
* Digital Research                          * Hisoft   GST  Metacomco  Seka
   move.l  #0,-(a7)    * GEM return        *        .        .        .
   trap    #1          *                   *        .        .        .
*
   rts                 * Return            *        .        .        .
*
   data                                    *  xxx  section d1       .  xxx .
   even                                    *  xxx  .  xxx  .  xxx  .  xxx .
*
*     VT52 screen location character equivalents
*
*   !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  0  1  2  3  4  5  6
* 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22
*
* 7  8  9  :  ;  <  =  >  ?  @  A  B  C  D  E  F  G  H  I  J
*23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42
*
*  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]
* 43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,590,60,61
*
*  ^  _  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
* 62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79
*
*     Data string to print (See Appendix C for codes)
*
*                                         *        .        .        .
*                                         *     section d2       .  even .
prdat:dc.b  50          * Length of text -1
      dc.b  27,'E'      * Clear screen
      dc.b  27,'b',0    * foregnd col white
      dc.b  27,'c',1    * backgnd col black
      dc.b  27,'Y  0,0' * Set cursor at 0,0
      dc.b  27,'Y!!1,1' * Set cursor at 1,1
      dc.b  27,'b',1    * foregnd col black
      dc.b  27,'c',0    * backgnd col white
      dc.b  27,'Y8f24,69' * Set curs @ 24,69
      dc.b  27,'p'      * Inverse video
      dc.b  27,'I'      * Up 1 line
      dc.b  'hi !'      * Say something witty
      dc.b  27,'Y,4'    * set cursor at 12,20
      dc.b  27,'q'      * Reset video
*
```

```
                                        * Hisoft   GST  Metacomco  Seka
* Digital Research

* Print null terminated string - uses ASCII & control codes
*                                                                  even .
*                                                                  .
*
mess: dc.b  'hello'      * Alternative print
      dc.b  10,10,7      * method for screen
      dc.b  'hello'      * printing.
      dc.b  10           *   Text
      dc.b  'hello'      *    linefeed
      dc.b  10,13        *     carriage return
      dc.b  'hello'      * and a bell
      dc.b  0            * End of string
*                                                                  even .
*
*

   end
```


SOUND DEMONSTRATION PROGRAM

This program provides a basic introduction to 'sound' programming on the Atari ST; where experimentation with each of the sounds provided is perhaps the best approach to understanding the effects of each argument.

Take care of the following general points:

Userstack Make sure it is large enough. It grows down in memory and it can overwrite the data area.

Timing It is necessary to provide a delay before an exit back to GEMDOS, TOS could reallocate the sound data bytes space.

```
* Digital Research                          * Hisoft   GST  Metacomco  Seka
*
*    Experimental TOS sounds program
*
   text                                     *  xxx  section c        .  code .
*
   move.l  a7,a5        * create            *
   move.l  #ustk,a7     * space for         *
   move.l  4(a5),a5     * program           *
   move.l  $c(a5),d0                        *
   add.l   $14(a5),d0                       *
   add.l   $1c(a5),d0                       *
   add.l   #$100,d0                         *
   move.l  d0,-(sp)                         *
   move.l  a5,-(sp)                         *
   move    d0,-(sp)                         *
   move    #$4a,-(sp)                       *
   trap    #1                               *
   add.l   #$c,sp                           *
*
start:
   move.l  #sound1,a1   *                   *
   jsr     dosound      *                   *
*
   move.l  #150,d1      * 15 secs           *
loopo:
   moveq   #-1,d2       * wait for          *
loopi:
   dbra    d2,loopi     * finish            *
   dbra    d1,loopo                         *
*
exit:
   clr.l   -(sp)        * GEMDOS ret        *
   trap    #$1                              *
   rts                                      *
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

dosound:
  move.l  a1,-(sp)     * sound pointer *
  move.w  #32,-(sp)    *               *
  trap    #14          *               *
  add.w   #6,sp        *               *
  rts                                  *

                                       * xxx  section d1      . data .
  bss                                  * xxx  .  xxx  .  xxx  . xxx  .
  even
*                                      *                      . blk.l .
     ds.l    64   * Large enough not   *                      . blk.l .
ustk: ds.l    1   * to overwrite data  *
*
                                       * xxx  section d2      . code .
  data                                 * xxx  .  xxx  .  xxx  . xxx  .
  even
*                    *          Bell
sound1:
*
        dc.b   0,$34 *\ chan A          *
        dc.b   1,0   */ 2150 hz         *
        dc.b   2,0   * \ chan           *
        dc.b   3,0   * /   B            *
        dc.b   4,0   *\ chan            *
        dc.b   5,0   */   C             *
        dc.b   6,0   * noise            *
        dc.b   7,$fe * enable A only    *
        dc.b   8,$10 * enable A envelop*
        dc.b   9,0   *  B off           *
        dc.b  10,0   *  C off           *
        dc.b  11,0   *\Single attack    *
        dc.b  12,$10 **envelope shape   *
        dc.b  13,9   */  1 0 0 1        *
        dc.b 130,100 *  delay           *
*
```



```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

* sound2:                 *          Siren
*
        dc.b   0,$fe *\ chan A          *
        dc.b   1,0   */ 440 hz  Hi note*
        dc.b   2,0   * \ chan           *
        dc.b   3,0   * /   B            *
        dc.b   4,0   *\ chan            *
        dc.b   5,0   */   C             *
        dc.b   6,0   * noise            *
        dc.b   7,$fe * enable A only    *
        dc.b   8,11  * A amplitude      *
        dc.b   9,0   *  B off           *
        dc.b  10,0   *  C off           *
        dc.b  11,0   *\   no            *
        dc.b  12,0   * envelope         *
        dc.b  13,0   */  shape          *
        dc.b 130,20  *                  *
*
        dc.b   0,$56 *\ chan A  Lo note*
        dc.b   1,1   */ 187 hz          *
        dc.b 130,20  *                  *
*
        dc.b   0,$fe,1,0,130,20 *Hinote*
        dc.b   0,$56,1,1,130,20 *Lonote*
*
* sound                   *          silence
*
        dc.b   8,0,9,0 * A & B off      *
        dc.b 130,50  *                  *
*
* sound3:                 *          gunshot
*
        dc.b   0,0,1,0,2,0,3,0,4,0,5,0 *
        dc.b   6,15  * medium noise period
        dc.b   7,199 * enable noise chans A,B & C
        dc.b   8,16  * \  using         *
        dc.b   9,16  *   envelope       *
        dc.b  10,16  * / control        *
        dc.b  11,0   *\ envelope period*
        dc.b  12,16  */                 *
        dc.b  13,0   * one cycle decay *
        dc.b 130,25  *                  *
*
* sound                   *          silence
*
        dc.b   8,0,9,0 * A & B off      *
        dc.b 130,50  *                  *
*
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

* sound4:              *        explosion
*
        dc.b   0,0,1,0,2,0,3,0,4,0,5,0 *
        dc.b   6,10  *  noise period   *
        dc.b   7,199 * enable noise chans A,B & C
        dc.b   8,16  * \  using        *
        dc.b   9,16  *   envelope      *
        dc.b  10,16  * / control       *
        dc.b  11,0   *\ envelope period*
        dc.b  12,80  */                *
        dc.b  13,0   * one cycle decay *
        dc.b 130,120 *                 *
*
* sound                *        silence
*
        dc.b   8,0,9,0,10,0 * A B & C off
        dc.b 130,100 *                 *
*
* sound5:              *        whistle
*
        dc.b   0,0,1,0,2,0,3,0,4,0,5,0,6,0
        dc.b   7,254 * enable tone A only
        dc.b   8,15  *                 *
        dc.b   9,0,10,0,11,0,12,0,13,0 *
        dc.b 128,60  * Initial tempreg *
        dc.b 129,0,-2,40 * reg-step-end*
        dc.b 130,2   *                 *
*
* exit list
*
        dc.b   7,255,8,0 * off         *
        dc.b 255,0   * return          *
*                                      *                    even
        end
```


A-Line

A-LINE PARAMETER TABLE

The following is the complete list of the A-line equates and functions. It may be used as a standard assembler header file to A-line programs.

```
*************************************************
*
*       A-line parameter table
*
*************************************************
*
V_CEL_HT        equ     -46 * .W Pixel cell height
V_CEL_MX        equ     -44 * .W Maximum cells across -1
V_CEL_MY        equ     -42 * .W Maximum cells high -1
V_CEL_WR        equ     -40 * .W Offset to next cell
V_COL_BG        equ     -38 * .W Background index color
V_COL_FG        equ     -36 * .W Foreground index color
V_CUR_AD        equ     -34 * .L Current cursor address
V_CUR_OFF       equ     -30 * .W Offset to 1st cell
V_CUR_CX        equ     -28 * .W X cursor position
V_CUR_CY        equ     -26 * .W Y cursor position
V_CUR_CNT       equ     -24 * .B Cursor flash interval
V_CUR_TIM       equ     -23 * .B Cursor countdown timer
V_FNT_AD        equ     -22 * .L Font address
V_FNT_ND        equ     -18 * .W Last font ASCII code
V_FNT_ST        equ     -16 * .W 1st font ASCII code
V_FNT_WR        equ     -14 * .W Font width
V_X_MAX         equ     -12 * .W Maximum X pixel screen value
V_OFF_AD        equ     -10 * .L Font offset table address
V_STATUS        equ     -6  * .W Text status byte
V_Y_MAX         equ     -4  * .W Maximum Y pixel screen value
*
VPLANES         equ     0   * .W # video planes
VWRAP           equ     2   * .W # bytes/video
CONTRL          equ     4   * .L \
INTIN           equ     8   * .L
PTSIN           equ     12  * .L    array pointers
INTOUT          equ     16  * .L
PTSOUT          equ     20  * .L /
COLBIT0         equ     24  * .W \ 1 * )
COLBIT1         equ     26  * .W * 2 * ) write
COLBIT2         equ     28  * .W * 4 * ) color
COLBIT3         equ     30  * .W / 8 * )
LSTLIN          equ     32  * .W -1
```

```
LNMASK      equ   34  * .W VDI line style
WMODE       equ   36  * .W Write mode
*
X1          equ   38  * .W \
Y1          equ   40  * .W   coordinates
X2          equ   42  * .W
X2          equ   44  * .W /
*
PATPTR      equ   46  * .L Current fill pattern pointer
PATMSK      equ   50  * .W Length of fill pattern mask
MFILL       equ   52  * .W 0_single plane
CLIP        equ   54  * .W 0_no clipping
*
XMINCL      equ   56  * .W \
YMINCL      equ   58  * .W   Clipping
XMAXCL      equ   60  * .W     values
YMAXCL      equ   62  * .W /
XDDA        equ   64  * .W txtblt x dda accumulator
DDAINC      equ   66  * .W txtblt scale factor
SCALDIR     equ   68  * .W 0_down
*
MONO        equ   70  * .W 0_font monospaced
SRCX        equ   72  * .W \ Coordinates of character
SRCY        equ   74  * .W / in font form
DESTX       equ   76  * .W \ Coordinates of character
DESTY       equ   78  * .W / on screen
DELX        equ   80  * .W Charachter width
DELY        equ   82  * .W Charachter height
*
FBASE       equ   84  * .L Font form pointer
FWIDTH      equ   88  * .W   width
STYLE       equ   90  * .W   style
LITEMSK     equ   92  * .W Lighten text mask
SKEWMSK     equ   94  * .W Skew text mask
WEIGHT      equ   96  * .W Extra text width
ROFF        equ   98  * .W High offset skew
LOFF        equ   100 * .W Low offset skew
SCALE       equ   102 * .W 0_no scaling
CHUP        equ   104 * .W 0_horizontal orientation
TEXTFG      equ   106 * .W Text foreground color
SCRTCHP     equ   108 * .L Text effects buffer
SCRPT2      equ   112 * .W Offset to scale buffer
TEXTBG      equ   114 * .W Text background color
COPYTRAN    equ   116 * .W Copy raster type flag
SEEDABORT   equ   118 * .W Abort fill routine pointer
*
```



```
*************************************************
*
*          A-line function calls
*
*************************************************
*
init        equ   $a000
putpix      equ   init+1  * Put pixel
getpix      equ   init+2  * Get pixel
abline      equ   init+3  * Draw a line
habline     equ   init+4  * Horizontal line
rectfill    equ   init+5  * Draw filled rectangle
polyfill    equ   init+6  * Draw 1 line polygon fill
bitblt      equ   init+7  * Bit block transfer
textblt     equ   init+8  * Text block transfer
showcur     equ   init+9  * Show mouse
hidecur     equ   init+10 * Hide mouse
chgcur      equ   init+11 * Transform mouse form
unsprite    equ   init+12 * Undraw previous sprit
drsprite    equ   init+13 * Draw sprite
copyrstr    equ   init+14 * Copy raster form
seedfill    equ   init+15 * Polygon fill
```

SPRITE DEMONSTRATION

The following A-line program is deliberately compressed to show the small number of lines of assembler used to control sprites. The program produces an alternate black and white sprite crossing a monochrome screen.

```
* Digital Research                          * Hisoft   GST  Metacomco  Seka
*
init          equ     $a000   * initialize .         .         init:    .
unsprite      equ     init+12 * undraw sprite .      .         unsprite:
drsprite      equ     init+13 * draw sprite .        .         drsprite:
*
V_X_MAX       equ     -12     * Max X pixel scrn val .         V_X_MAX:
*
  text                        *  xxx  section c                . code   .
*
start: clr.l   -(sp)        *  Set     *  .  .  .  .  .  .  .  .  .  .  *1
  move.w  #$20,-(sp)        * super    *  .  .  .  .  .  .  .  .  .  .  *1
  trap    #1                * mode     *  .  .  .  .  .  .  .  .  .  .  *1
  addq.l  #6,sp             *          *  .  .  .  .  .  .  .  .  .  .  *1
  move.l  d0,stksv          * save stack*  .  .  .  .  .  .  .  .  .  .  *1
  move.w  #0,olda           * versn flag*  .  .  .  .  .  .  .  .  .  .  *2
  move.l  #0,a2             *          *  .  .  .  .  .  .  .  .  .  .  *2
  dc.w    init              *   Init   *  .  .  .  .  .  .  .  .  .  .
  move.l  a2,d2             *   aline  *  .  .  .  .  .  .  .  .  .  .  *2
  bne     a2ok              * registers *  .  .  .  .  .  .  .  .  .  .  *2
  lea     #-60(a1),a2       *          *<------ -60(a1),a2 -------->  .  *2
  move.w  #-1,olda          *          *  .  .  .  .  .  .  .  .  .  .  *2
*
a2ok:  move.l  $34(a2),a3   * draw addr (4*13)                            *2
  move.w  #V_X_MAX(a0),a5   * get max width <---- V_X_MAX(a0),a5 -----> .
  move.w  #0,d0             * init x   *  .  .  .  .  .  .  .  .  .  .
  move.w  #50,d1            * init y   *  .  .  .  .  .  .  .  .  .  .
  move.w  #10,d2            * scan count*  .  .  .  .  .  .  .  .  .  .
  lea     sprit,a0          * sprite add*  .  .  .  .  .  .  .  .  .  .
  lea     save,a2           * bg savearea  .  .  .  .  .  .  .  .  .  .
  movea.l a0,a4             * sprite col*  .  .  .  .  .  .  .  .  .  .
  adda.l  #6,a4             *  pointer *  .  .  .  .  .  .  .  .  .  .
*
setcol: move.w  (a4),d3     * get color *  .  .  .  .  .  .  .  .  .  .
  bne     white             *          *  .  .  .  .  .  .  .  .  .  .
  move.l  #$00010001,(a4)   *  black   *  .  .  .  .  .  .  .  .  .  .
  bra     loop              *   color  *  .  .  .  .  .  .  .  .  .  .
white: move.l  #0,(a4)      * set white *  .  .  .  .  .  .  .  .  .  .
loop:  movem.l d0-d3/a0-a3,-(sp) *sav r*  .  .  .  .  .  .  .  .  .  .
  tst.w   olda              * test versn*  .  .  .  .  .  .  .  .  .  *2
  beq     new               *          *  .  .  .  .  .  .  .  .  .  *2
  jsr     (a3)              * old versn *  .  .  .  .  .  .  .  .  .  *2
  bra     cont              *          *  .  .  .  .  .  .  .  .  .  *2
```



```
* Digital Research                          * Hisoft   GST  Metacomco  Seka

new:   dc.w    drsprite     * new versn *  .  .  .  .  .  .  .  .  .  .
cont:  move.w  #2000,d0     *          *  .  .  .  .  .  .  .  .  .  *3
wait:  dbra    d0,wait      * delay    *  .  .  .  .  .  .  .  .  .  *3
  lea      save,a2          * bg savarea*  .  .  .  .  .  .  .  .  .  .
  dc.w     unsprite         *          *  .  .  .  .  .  .  .  .  .  .
  movem.l (sp)+,d0-d3/a0-a3 * unsave r*  .  .  .  .  .  .  .  .  .  .
  add.w    #1,d0            * slide over*  .  .  .  .  .  .  .  .  .  .
  cmp.w    a5,d0            *   screen *  .  .  .  .  .  .  .  .  .  .
  ble      loop             *          *  .  .  .  .  .  .  .  .  .  .
  move.w   #0,d0            * init x   *  .  .  .  .  .  .  .  .  .  .
  add.w    #10,d1           * drop y   *  .  .  .  .  .  .  .  .  .  .
*
  sub.w    #1,d2            * count down*  .  .  .  .  .  .  .  .  .  .
  bne      setcol           * and again *  .  .  .  .  .  .  .  .  .  .
  move.l   stksv,-(sp)      * back     *  .  .  .  .  .  .  .  .  .  *1
  move.w   #$20,-(sp)       *  to      *  .  .  .  .  .  .  .  .  .  *1
  trap     #1               * user     *  .  .  .  .  .  .  .  .  .  *1
  addq.l   #6,sp            * mode     *  .  .  .  .  .  .  .  .  .  *1
  move.w   #0,-(sp)         * back     *  .  .  .  .  .  .  .  .  .  .
  trap     #1               * to GEM   *  .  .  .  .  .  .  .  .  .  .
*
  data                                 *  xxx  section d       .  xxx  .
  even                                 *  xxx  .  xxx  .  xxx  .  xxx  .
*
sprit: dc.w    0,0          * x,y      *
       dc.w    -1           * 1_vdi, -1_xor
       dc.w    0            * bg col   *
       dc.w    0            * fg col   *
ghoul: dc.w    $ffff                   *
       dc.w    $03c0                   *
       dc.w    $ffff                   *
       dc.w    $0ff0                   *
       dc.w    $ffff                   *
       dc.w    $1ff8                   *
       dc.w    $ffff                   *
       dc.w    $3ffc                   *
       dc.w    $ffff                   *
       dc.w    $73ce                   *
       dc.w    $ffff                   *
       dc.w    $73ce                   *
       dc.w    $ffff                   *
       dc.w    $ffff                   *
       dc.w    $ffff                   *
       dc.w    $ffff                   *
       dc.w    $ffff                   *
       dc.w    $fbdf                   *
       dc.w    $ffff                   *
       dc.w    $f81f                   *
```

```
* Digital Research                    * Hisoft   GST  Metacomco  Seka

        dc.w    $ffff                 *    .     .     .     .
        dc.w    $ffff                 *    .     .     .     .
        dc.w    $ffff                 *    .     .     .     .
        dc.w    $67e6                 *    .     .     .     .
        dc.w    $ffff                 *    .     .     .     .
        dc.w    $300c                 *    .     .     .     .
        dc.w    $ffff                 *    .     .     .     .
        dc.w    $1ff8                 *    .     .     .     .
        dc.w    $ffff                 *    .     .     .     .
        dc.w    $0420                 *    .     .     .     .
        dc.w    $ffff                 *    .     .     .     .
        dc.w    $1818                 *    .     .     .     .
*
  bss                                 *  xxx  section d     .  data .
  even                                *  xxx  .  xxx  .  xxx  .  xxx  .
*
stksv: ds.l    1                      *    .     .     .  blk.l .   *1
save:  ds.b    74                     *    .     .     .  blk.b .
olda:  ds.w    1                      *    .     .     .  blk.w .   *2
*
  end
```

*1 There is no requirement to run this program in supervisor mode, these lines of code may be ommitted.

*2 Some versions of the disk based TOS incorrectly return the value of A2. These lines of code are not requred by ROM based versions of the ST.

*3 The use of the following code provides more stable sprites +

```
MOVE   #37,-(sp)      * wait for Vblank
TRAP   #14            * XBIOS call
ADDQ   #2,sp          * tidy stack
```

The programmer might also contemplate hiding the busy-bee cursor.

Appendix M – Glossary

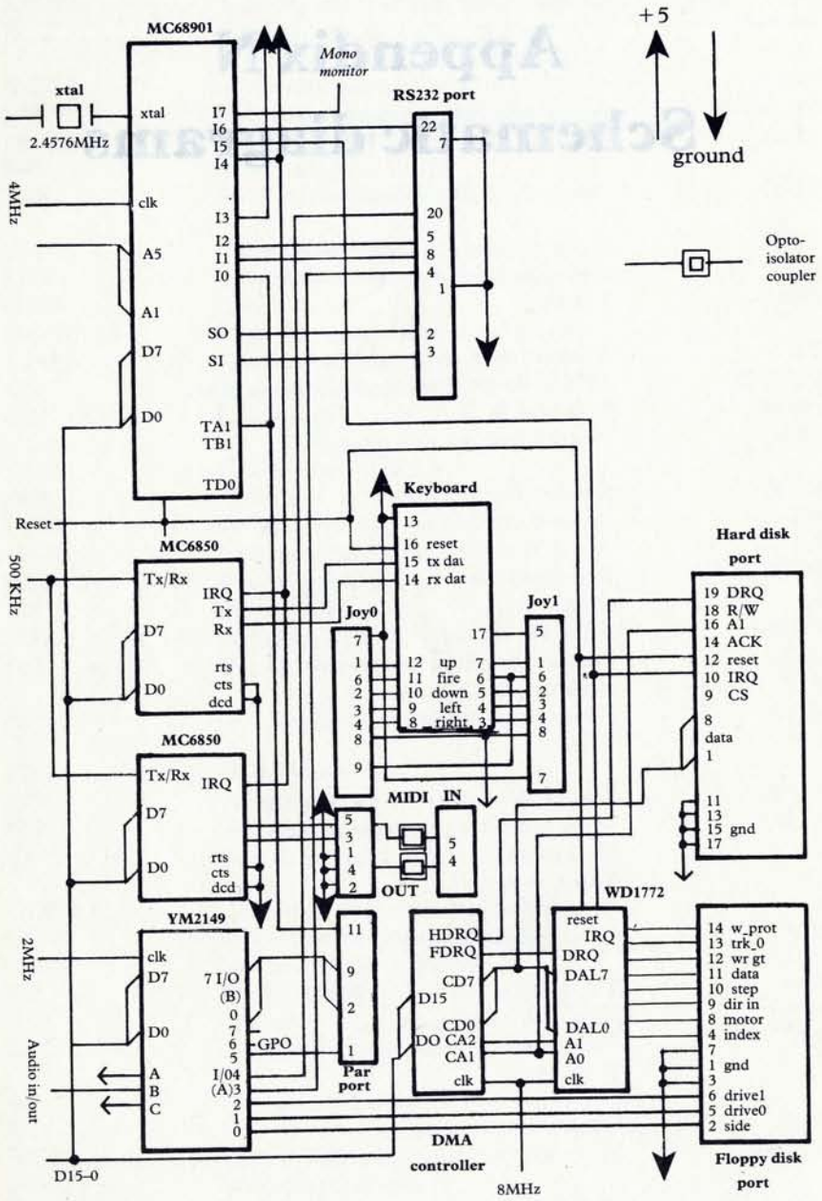
ADE	ASCII decimal equivalent
AES	Application environment services
ACIA	Asynchronous communications interface adaptor
ANSI	American national standards institute
ASCII	American standard code for information interchange
AUX	Auxilary
BCD	Binary coded decimal
BDOS	Basic disk operating system
BIOS	Basic input/output system
BPB	BIOS parameter block
BSS	Block storage segment
CCP	Console command processor
CCR	Condition code register
CON	Console
CP/M	Control program for microcomputers
CPU	Central processing unit
CRC	Cyclic redundancy check
CTS	Clear to send
DCD	Data carrier detect
DIR	Directory
DMA	Direct memory access
DOS	Disk operating system
DPB	Disk parameter block
DS	Double sided
DTR	Data terminal ready
D/A	Digital to analogue
EPB	Exception parameter block
FAT	File allocation table
FCB	File control block
FDC	Floppy disk controller
FIFO	First in first out register
GDOS	Graphics device operating system
GEM	Graphics environment manager
GIOS	Graphics input/output system
GP	General purpose
Grd	Ground
GSX	Graphic system extension
HDC	Hard disk controller
ID	Identification
ikbd	Intelligent keyboard
IPL	Interrupt level
I/O	Input/output

LPB	Load parameter block
LSB	Least significant byte/bit
LST	List
MD	Memory descriptor
MFDB	Memory form definition block
MFP	Multi function peripheral
MIDI	Musical instruments digital interface
MS-DOS	Microsoft disk operating system
MSB	Most significant bit
NDC	Normalized device coordinates
OEM	Other equipment manufacturer
OS	Operating System
OSC	Oscillator
PC	Program counter
PC-DOS	IBM personal computer operating system
pk-pk	Peak to peak
PSG	Programmable sound generator
RAM	Random access memory
RC	Raster coordinate
RF	Radio frequency
RGB	Red–green–blue
Ri	Ring
ROM	Read only memory
RSX	Resident system extension
RTE	Return from exception
RTS	Return form subroutine
Rx	Receive
SASI	Shugart associates standard interface
SCSI	Small computer systems interface
SP	Stack pointer
SR	Status register
SS	Single sided
SSP	Supervisor stack pointer
TOS	The operating system
TPA	Transient program area
TTL	Transistor-transistor logic
Tx	Transmit
ULA	Uncommitted logic array
USART	Universal synchronous/asynchronous receiver/transmitter
USP	User stack pointer

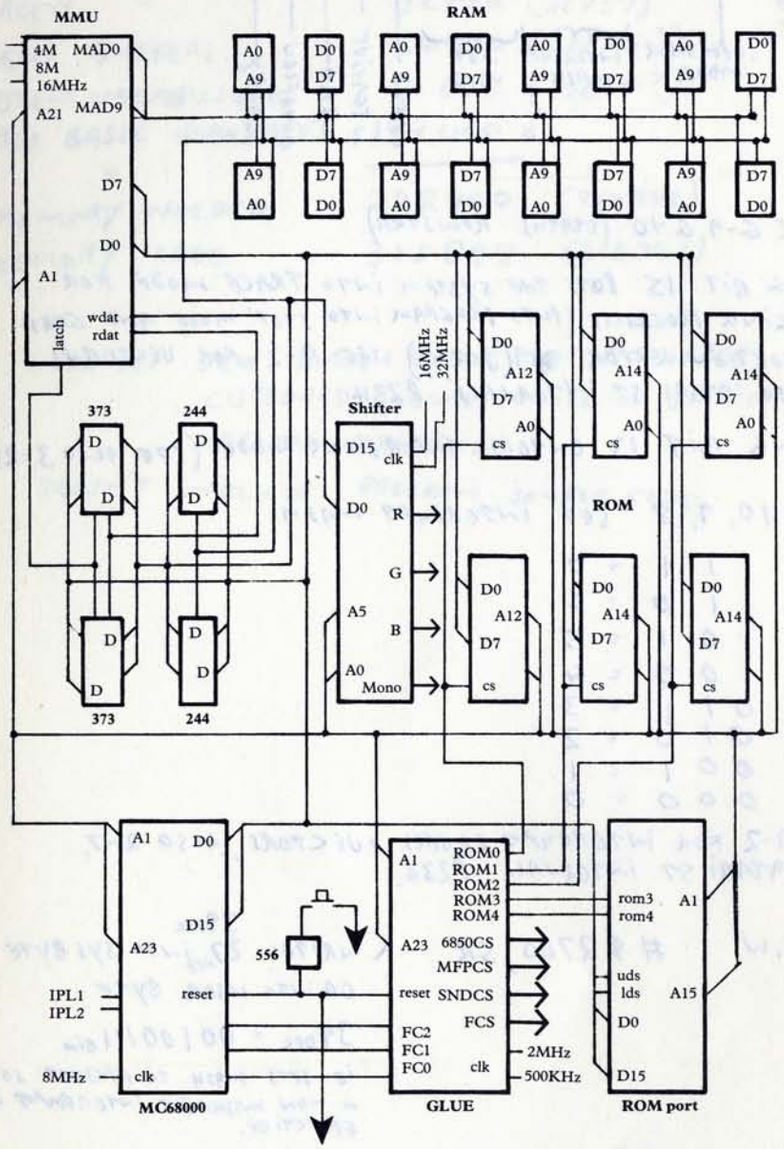
VBI	Vertical blank interrupt
VDI	Virtual device interface
VLSI	Very large scale integration

Appendix N
Schematic diagrams

ATARI ST peripheral control diagram



ATARI ST schematic diagram



STATUS REGISTER

SYSTEM BYTE								USER BYTE							
15	X	13	X	X	10	9	8	X	X	X	4	3	2	1	0
TRACE MODE		SUPERVISOR MODE			INTERRUPT MASK			NOT USED			EXTEND	NEGATIVE	ZERO	OVERFLOW	CARRY

SEE G-9, G-10 (STATUS REGISTER)

SETTING BIT 15 PUTS THE SYSTEM INTO TRACE MODE FOR DEBUGGING PROGRAMS. (PUTS PROGRAM INTO 'STEP' MODE AND GOES TO EXCEPTION VECTOR 9 (#024)) SEE A-2 FOR VECTORS
SEE ALSO 'ATARI ST INTERNALS' P234

SETTING BIT 13 ENTERS SUPERVISOR MODE (SEE ALSO 3-21)

BITS 10, 9, 8 SET INTERRUPT MASK

1	1	1	= 7
1	1	0	= 6
1	0	1	= 5
1	0	0	= 4
0	1	1	= 3
0	1	0	= 2
0	0	1	= 1
0	0	0	= 0

SEE A-2 FOR INTERRUPT LEVELS + VECTORS, ALSO 2-7,
ALSO 'ATARI ST INTERNALS' P236.

MOVE.W #$2700, SR — WRITES 27_{HEX} (39 DEC) INTO SYS BYTE AND 00 INTO USER BYTE.
39 DEC = 00100111 BIN
IE SETS MASK TO HIGHEST SO ONLY A NON MASKABLE INTERRUPT WILL BE EFFECTIVE.

#$2500, SR — 37 DEC = 00100101 BIN
ONLY NMI AND MFP INTERRUPTS.

#$2400, SR — 36 DEC = 00100100 BIN
NMI, MFP AND VB SYNC

#$2000, SR — ALL INTERRUPTS EFFECTIVE, CAN ALSO BE USED TO ENTER SUPERVISOR MODE (?)

MEMORY ALLOCATION UNDER FAST BASIC

START MEMORY	524288	
SCREEN "	32000	(32767)
GEM BUFFERS	64000	(65535 ?)
SYSTEM VARIABLES ETC	10000	(10240 ?)
FAST BASIC WORKSPACE	102400 *	
MEMORY NEEDED	208400	(210942)
MEMORY FREE	315888	(313346)

* INCLUDES 32K SCREEN BUFFER (FOR PUT + GRAB)
CLIPBOARD (EQUAL TO SIZE OF SOURCE CODE)
VARIOUS VARIABLES
DOESN'T INCLUDE PROGRAM SOURCE CODE